UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

Desarrollo de una aplicación de Podcast sobre Android

Autor: Carlos Salguero Guzmán
Tutor: Celeste Campo

18 de agosto de 2011

Agradecimientos

Muchos recuerdos vienen a mi cabeza a la finalización de este proyecto de fin de carrera. Momentos buenos y también momentos malos. No ha sido una tarea sencilla el poder terminarlo, pero ahora que llega su fin no puedo olvidarme de todas aquellas personas que me han apoyado en los buenos y malos momentos de los últimos años y que me han facilitado de una u otra manera este difícil objetivo. A todos ellos, amigos y familiares, muchísimas gracias.

También me gustaría dar mi más sincero agradecimiento a Celeste, mi tutora, por su tiempo y dedicación durante todos estos meses.

Finalmente, no puedo olvidarme de mis padres, Eugenio y Paquita, y de Alberto, Cristina, Uge y Maribel, por su incesante apoyo, las risas compartidas con ellos y las magníficas discusiones también. Y por supuesto a Raquel, porque siempre está ahí y porque con ella no puede haber malos momentos.

Carlos

Resumen

Hoy en día, los dispositivos móviles y, más concretamente, los teléfonos móviles de última generación, constituyen una realidad que ofrece al usuario no sólo una forma de comunicarse, sino también de divertirse, aprender e interactuar con aquello que le resulte interesante. El sistema operativo Android, propiedad de Google, busca ser una firme alternativa a otros sistemas ya ampliamente extendidos como Symbian o IOS. Por otra parte, la evolución de Internet hasta desembocar en lo que se denomina la web 2.0, ha supuesto una transición hacia aplicaciones y tecnologías cada vez más orientadas al usuario. Este es el caso de los podcasts y RSS.

El presente proyecto busca aunar las posibilidades que ofrecen ambas tecnologías en expansión para sacar un mayor beneficio del que cada una de ellas ofrece por separado. Por un lado, reduce las limitaciones de las suscripciones tradicionales a podcasts, expandiendo además las posibilidades del teléfono móvil. Para tal efecto, abarca el desarrollo completo de la aplicación de nombre DroidCatcher, que permite realizar una gestión completa de suscripciones a Poscast desde el propio teléfono móvil y que busca servir de manual práctico para toda aquella persona que esté interesada en el desarrollo de aplicaciones para Android.

Abstract

Nowadays, mobile devices and, concretly, smart phones are becoming a reality that offers users not only a way of communication among them, but also a way of having fun, learning and interacting with those things they consider interesting. Android operative system, owned by Google, aims to be a solid alternative to other widely extended systems like Symbian or IOS. Otherwise, Internet evolution has led to what is called web 2.0, that has resulted in a transition to user-oriented applications and technologies. This is the case of podcasts and RSS.

This project seeks to join the possibilities that both technologies offer to get a bigger benefit than each one provides separately. First, it reduces limitations of traditional podcasts subscription and it also expands the capabilities of current smart phones. For this purpose, this project covers the complete development of the application named DroidCatcher, which allows the management of podcasts subscriptions from the mobile phone and seeks to serve as a practical guide for anyone who is interested in applications development over Android platform.

Índice general

Índice de figuras

Capítulo 1

Introducción

En las siguientes líneas se hace una breve introducción al presente proyecto, exponiendo cuál es su motivación, qué objetivos son los que persigue y cuáles son los contenidos ofrecidos en esta memoria.

1.1. Motivación del proyecto

Hoy en día, las comunicaciones móviles están tan integradas en nuestro mundo que la mayoría de la gente se siente incómoda sin un teléfono móvil. Ayer, las funciones más populares de un teléfono eran las llamadas y los mensajes de texto (SMS). Hoy, sin embargo, las prioridades apuntan a direcciones tan variadas como distintas a aquella realidad. Un teléfono moderno, habitualmente denominado *SmartPhone* [1], es un dispositivo multifuncional que ayuda a la gente no sólo a comunicarse entre ella sino también a aprender, informarse y divertirse. Y todo esto ha sido posible gracias al desarrollo de la tecnología y, particularmente, al desarrollo de aplicaciones móviles que se benefician de ella [2].

Las primeras aplicaciones móviles instaladas en teléfonos datan de finales de la década de los 90. Básicamente, se trataba de aplicaciones multimedia muy sencillas: pequeños juegos arcade, calculadoras, pequeños calendarios, etc. Con la llegada de la tecnología WAP [3] y posteriormente nuevas tecnologías para la transmisión de datos (GPRS [4], EDGE [5], 3G [6]), el desarrollo de teléfonos móviles se convirtió en algo más fácil y rápido. De este modo, el inicio del nuevo milenio fue testigo de una rápida evolución del mercado de los contenidos móviles en general y de las aplicaciones para éstos en particular.

Para entonces, el mercado de dispositivos móviles estaba siendo gradualmente conquistado por los denominados SmartPhones o teléfonos inteligen-

tes. Con muchas más características y mejor rendimiento que los teléfonos móviles ordinarios, se diferenciaban también de éstos últimos en la variedad de sistemas operativos disponibles para ellos. Sistemas operativos tales como Windows Mobile [7], Symbian [8], RIM [9] y, más cerca de nuestros días, IOS [10] y Android. Este hecho abría la posibilidad al desarrollo de aplicaciones por parte de terceros, en contraste con el entorno de desarrollo más convencional de los teléfonos móviles *ordinarios*.

Esto llevó a un increíblemente mayor ritmo de desarrollo de teléfonos celulares. Los fabricantes aspiraban a hacer de sus productos los más atractivos para los usuarios, introduciendo para ello más y más aplicaciones.

Aparte de la cantidad, la calidad importa también. Se hizo evidente que el desarrollo de los teléfonos móviles tenía que ser fácil e intuitivo. Cada compañía trata de facilitar el proceso de desarrollo, de modo que el usuario medio sea capaz de personalizar sus dispositivos.

El volumen de mercado de las aplicaciones móviles es de muchos millones de euros [11].

Por todo lo expuesto anteriormente, es fácil comprender el interés y la importancia del desarrollo de aplicaciones móviles, por lo que este proyecto de fin de carrera tratará de acercar al lector el proceso de crear una aplicación sobre una plataforma con un futuro bastante prometedor, como es el sistema operativo Android [12].

Existe una amplia variedad de categorías de aplicaciones: desde herramientas productivas, educacionales, fotográficas, videojuegos y, por supuesto, multimedia. ¿Por qué hacer una aplicación sobre Podcast? Los Podcast son cada vez más un instrumento, sino fundamental, si muy importante para la mayoría de empresas audiovisuales actuales. Cadenas de radio, televisiones, periódicos, revistas... todos ellos utilizan esta tecnología para ofrecer al usuario contenidos multimedia de modo que el usuario pueda acceder a él cuando pueda o quiera, sin restricciones de tiempo o situación geográfica. Una herramienta que permita gestionar estas suscripciones y sus archivos multimedia desde un teléfono móvil puede resultar de una gran utilidad y es por esto por lo que se ha decidido hacer esta aplicación.

1.2. Objetivos

Para dirigir con mayor éxito los esfuerzos en el desarrollo del proyecto, se establecen una serie de objetivos que abarquen las actividades que se pretenden realizar. De este modo, también se podrá evaluar al final del desarrollo el grado de cumplimiento y satisfacción alcanzado en el mismo.

El objetivo principal del proyecto es el desarrollo de una aplicación para

la gestión de suscripciones podcasts desde un teléfono móvil con sistema operativo Android. Como consecuencia de este desarrollo se cumplimentarán otros objetivos:

- Conocer las principales características de Android. El desarrollo de toda aplicación requiere ciertos conocimientos, tanto del lenguaje de programación utilizado como de la plataforma sobre la que se va a desarrollar. Este proyecto de fin de carrera pretende ilustrar de forma práctica, mediante el desarrollo de una aplicación, indagando primeramente en las características principales de Android para luego centrarse en el desarrollo sobre la misma. No se trata, por tanto, de un estudio en profundidad acerca del Sistema Operativo en sí, por lo que se requieren conocimientos previos de programación.

- Familiarización con las tecnologías Podcast. RSS y los archivos multimedia son conceptos que están muy implantados en la sociedad actual. La aplicación desarrollada hace uso de estas tecnologías y por tanto se hace necesario el conocimiento de las mismas.

- Técnicas de desarrollo con Android. Guía técnica para el desarrollo de aplicaciones sobre la plataforma Android. Se pretende que el documento creado sirva a futuros desarrolladores como guía sobre aspectos concretos del desarrollo sobre Android. Se verán aspectos teóricos aplicados a un desarrollo práctico.

Siendo un poco más concretos y a modo de resumen:

- Estudiar las tecnologías Podcast
- Estudiar el desarrollo de aplicaciones en Android.
- Diseñar la aplicación.
- Implementar la aplicación.
- Probar la funcionalidad de la aplicación
- Documentar el diseño, implementación y pruebas.

1.3. Contenido de la memoria

A continuación se explica brevemente el contenido de cada capítulo y anexo incluido en esta memoria.

En el capítulo 1, *Introducción*, se expone la motivación del presente proyecto, los objetivos que persigue, así como una visión general de los contenidos de la memoria.

El capítulo 2, *Estado del Arte*, ofrece una descripción general de algunos de los aspectos técnicos relacionados con este proyecto, haciendo especial hincapié en las características de Android y de sus componentes, así como la tecnología RSS

En el capítulo 3, titulado *DroidCatcher: Gestor de Podcast*, se exponen los objetivos y características principales de la aplicación, detallando de forma ilustrada el funcionamiento y casos de uso de la misma.

El capítulo 4, de nombre *Desarrollo e implementación de DroidCatcher*, describe de forma detallada el proceso de implementación de la aplicación para Android. Cuenta con ejemplos de implementación de aspectos concretos de la aplicación, con ejemplos de código incluidos.

El capítulo 5, *Pruebas*, donde se explica el plan de pruebas de la aplicación, el diseño de las mismas así como los resultados obtenidos.

Capítulo 6, *Historia del proyecto*, detalla el ritmo de realización del proyecto. Se especifica para cada tarea realizada durante el proyecto el tiempo dedicado a cada una de ellas.

El capítulo 7, *Conclusiones y trabajos futuros*, repasa los objetivos propuestos en el capítulo 1 y comenta el grado de satisfacción alcanzado así como también las dificultades encontradas durante la realización del proyecto. Además se proponen futuras mejoras de la aplicación y posibles nuevas funcionalidades.

Por último, se encuentran los apéndices A y B. El primero, llamado *Manual de instalación y desarrollo*, contiene una guía breve de iniciación en el desarrollo de aplicaciones sobre Android utilizando la plataforma Eclipse. El segundo de ellos, *Manual de referencia*, contiene un glosario de términos así como el listado de referencias.

Capítulo 2

Estado del arte

2.1. Introducción a la plataforma Android

Android es una solución completa de software de código libre para teléfonos y dispositivos móviles [12]. Es un paquete que engloba un sistema operativo, un "runtime"de ejecución basado en Java, un conjunto de librerías de bajo y medio nivel y un conjunto inicial de aplicaciones destinadas al usuario final. Se distribuye bajo una licencia Apache, versión 2, una licencia libre permisiva que permite la integración con soluciones de código propietario [13].

Android busca causar un gran impacto en la industria de la comunicación móvil, estableciendo una plataforma abierta que permita un acceso fácil a prácticamente todas las funcionalidades hardware de los dispositivos en los que esté instalado, así como proveyendo de serie a los desarrolladores con librerías que favorezcan la creación ágil y rápida de aplicaciones. Se ha hecho especial énfasis en que las aplicaciones creadas por terceros no tendrán ningún tipo de desventaja en cuanto a funcionalidad y acceso al dispositivo que las aplicaciones "nativas"que se distribuirán originalmente con Android.

2.1.1. Arquitectura

Android proporciona un paquete completo de software a todos los niveles:

- Un kernel linux que sirve como base de la pila de software y se encarga de las funciones más básicas del sistema: gestión de drivers, seguridad, comunicaciones, etc.
- Una capa de librerías de bajo nivel en C y C++, como SQLite para persistencia de datos; SGL, desarrollada por Skia, otra adquisición de

Google; OpenGL ES para gestión de gráficos 3D, con aceleración 3D opcional y Webkit como navegador web embebido y motor de renderizado HTML.

- Un framework para el desarrollo de aplicaciones, dividido en subsistemas para gestión del sistema como el "package manager"; gestión del hardware del teléfono anfitrión ("telephony manager") o acceso a APIs sofisticadas de geolocalización o mensajería XMPP.

- También incluye un sistema de vistas para manejar la interfaz de usuario de las aplicaciones, que incluyen posibilidad de visualización de mapas o renderizado HTML directamente en la interfaz gráfica de la aplicación.

- Un conjunto de aplicaciones (navegador, agenda, gestión del teléfono).

En la figura 2.1 se muestra la mayor parte de los componentes del sistema operativo Android.

2.1.2. Versiones de la plataforma

Se han realizado numerosas actualizaciones de la plataforma Android desde su lanzamiento original. Estas actualizaciones del sistema operativo normalmente solucionan algunos errores y añaden nuevas características. Generalmente, cada nueva versión de Android es desarrollada bajo un nombre en código basado en un postre o helado.

A continuación se explican muy brevemente las mejoras introducidas por las últimas actualizaciones de Android:

- **2.0/2.1 (Eclair)**: introdujo una renovada interfaz de usuario así como HTML5 y soporte a **Exchange ActiveSync 2.5** [14].

- **2.2 (Froyo)**: introdujo mejoras en la velocidad con la optimización JIT y el motor **Chrome V8 Javascript**. También añadía soporte para **Adobe Flash** [15] y puntos de acceso WI-FI [16].

- **2.3 (Gingerbread)**: redefine la interfaz de usuario, mejoraba capacidades de teclado y funciones como **copy/paste** y añade soporte para **NFC (Near Field Communication)** [17].

- **3.0 (Honeycomb)**: actualización orientada a *tablets*. Soporta dispositivos con tamaños de pantalla mayor e introduce muchas características para la interfaz de usuario nuevas. Además, soporta procesadores multi

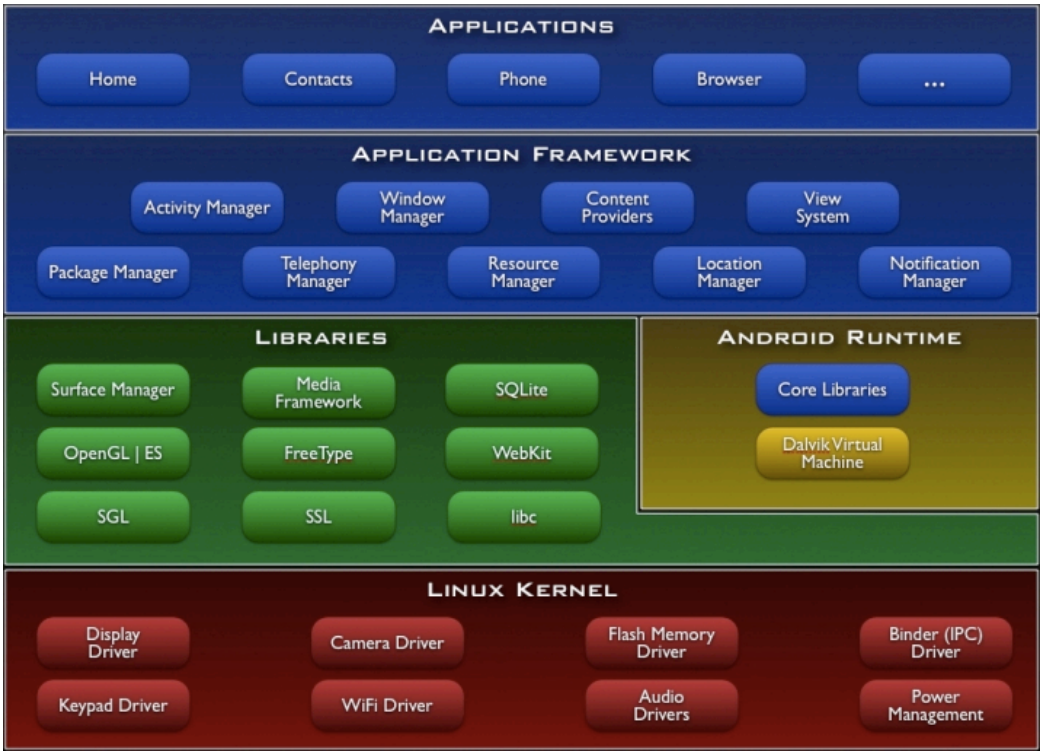

Figura 2.1: Arquitectura de Android.

Fuente: Google Inc.

núcleo y aceleración hardware para los gráficos. En esta ocasión, Google decidió no poner a disposición de la gente el código fuente de esta versión, al menos hasta que no estuviera "listo", en palabras de Andy Rubin. Finalmente, Google anunció que el código fuente no saldría a la luz hasta una próxima actualización, 2.4, que fuese una mezcla de las versiones anteriores Gingerbread y Honeycomb [18].

- **2.4**: la futura versión de Android en aparecer. Se trata de una combinación de Gingerbread y Honeycomb. Será lanzada en el último trimestre de 2011 [19].

NOTA: la aplicación DroidCatcher ha sido desarrollada y probada utilizando la versión 2.2 (API nivel 8) de Android. La aplicación no puede ser instalada en versiones anteriores del sistema.

2.1.3. La máquina virtual Dalvik

En Android, todas las aplicaciones se programan en el lenguaje **Java** y se ejecutan mediante una máquina virtual (VM) de nombre Dalvik, específicamente diseñada para Android [20]. Esta máquina virtual ha sido optimizada y adaptada a las peculiaridades propias de los dispositivos móviles (menor capacidad de proceso, baja memoria, alimentación por batería, etc.) y trabaja con ficheros de extensión *.dex* (DalvikExecutables). Dalvik no trabaja directamente con el *bytecode* de Java, sino que lo transforma en un código más eficiente que el original, pensado para procesadores pequeños.

Gracias a la herramienta "dx", esta transformación es posible: los ficheros *.class* de Java se compilan en ficheros *.dex*, de forma que cada uno de ellos puede contener varias clases. Después, este resultado se comprime en un único archivo de extensión *.apk* (Android Package), que es el que se distribuirá en el dispositivo móvil. Dalvik permite varias instancias simultáneas de la máquina virtual, y a diferencia de otras máquinas virtuales, está basada en registros y no en pila, lo que implica que las instrucciones son más reducidas y el número de accesos a memoria es menor.

La creación de una VM propia es un movimiento estratégico que permite a Google evitar conflictos con Sun por la licencia de la máquina virtual, así como asegurarse el poder innovar y modificar ésta sin tener que batallar dentro del JCP (Java Community Process).

Arquitectura

A diferencia de la mayoría de máquinas virtuales, (también Java VM), que están basadas en pila, la máquina virtual Dalvik **está basada en registro**.

Las ventajas de utilizar una arquitectura basada en pila frente a una basada en registro es objeto de debate. Generalmente, las máquinas virtuales basadas en pila usan instrucciones para cargar datos en la pila y manipularlos y, de este modo, se necesitan más instrucciones que en las máquinas virtuales basadas en registro para realizar el mismo código de alto nivel. Sin embargo, las instrucciones en una máquina virtual basada en registro deben contener tanto el registro origen como el destino por lo que generalmente son más largas. Esta diferencia es de una importancia básica para los intérpretes de las VM para quienes la ejecución de código máquina tiende a ser costoso junto con otros factores de igual relevancia como la compilación JIT (*Just in Time*) [21] .

La herramienta **dx** es usada para convertir algunos (no todos) ficheros *.class* de Java al formato .dex. Se pueden incluir varias clases java en un único fichero *.dex*. Las cadenas u otras constantes que estén duplicadas solamente serán incluidas una vez en el fichero *.dex* para ahorrar espacio. El *java bytecode* también es convertido a un set de instrucciones usado por Dalvik. Un fichero *.dex* sin comprimir suele ser más pequeño que un archivo java comprimido (*.jar*) proveniente del mismo fichero *.class*.

Los ejecutables del Dalvik pueden ser modificados de nuevo cuando son instalados en un dispositivo móvil para hacer cierta optimización.

Desde la versión 2.2 de Android, Dalvik tiene compilación **JIT** (*Just in Time*).

Al haber sido optimizada para requerimientos de baja memoria, Dalvik tiene algunas características especiales que lo diferencian del resto de máquinas virtuales:

- la máquina virtual se adelgazó para utilizar menos espacio
- el pool de constantes ha sido modificado para que use solamente índices de 32 bits, para simplificar el intérprete
- usa su propio *bytecode*, no el *bytecode* de JAVA

Además, Dalvik fue diseñada para que un dispositivo pudiera ejecutar múltiples instancias de la máquina virtual de forma eficiente.

2.1.4. Estructura de una aplicación Android

La estructura de una aplicación Android está definida por la interacción de distintos componentes, haciendo énfasis en la "agrupación débil"de distintas piezas. La aplicación hará uso de las distintas APIs expuestas por Android, de forma que los componentes encargados de realizar cada tarea

puedan ser manipulados o reemplazados sin problemas, asegurando la máxima flexibilidad. Por ejemplo, una aplicación puede permitir al usuario elegir fotos mediante el componente "Galería" o, por ejemplo, reemplazar esa "Galería"por una selección de fotos a través de un servicio online. Los principales componentes de una aplicación serían:

- Activity: representa cada una de las principales tareas que el usuario puede llevar a cabo en la aplicación. Típica (aunque no necesariamente) corresponderá a una pantalla específica de la aplicación y, también normalmente, una "activity"será el punto de entrada (pantalla inicial) de nuestra aplicación. Desde ella se invocarán las vistas, específicas o layouts, para la aplicación.

- IntentReceiver: permite a la aplicación declarar ciertos "callback"que responderán a cambios en el estado del terminal. P.ej. llamada o email recibido, cambio en la geolocalización, etc.

- Service: una tarea que corre en el background y que puede y debe ejecutarse sin interacción con el usuario. Una aplicación puede mandar los mensajes necesarios a un determinado servicio activo.

- ContentProvider: establece una capa que permite a las distintas aplicaciones compartir datos. Con independencia del almacenamiento local que utilicen para sus propósitos, las aplicaciones necesitan declarar ContentProviders para poner a disposición de otros procesos los datos que consideren necesarios.

Estas son algunas de las principales [22], pero no las únicas piezas de construcción de la aplicación. También es interesante que se defina como pieza de primer nivel el sistema de notificaciones en pantalla, que se recomienda como principal vía de comunicación con el usuario.

2.1.5. Fichero AndroidManifest

Toda aplicación Android ha de tener un fichero *AndroidManifest.xml* (con exactamente ese nombre) en su directorio raíz. El manifiesto presenta información esencial acerca de la aplicación al sistema Android, información que el sistema debe tener antes de que pueda ejecutar el código de cualquier aplicación. Entre otras cosas, el fichero hace lo siguiente:

- Le da nombre al paquete JAVA de la aplicación. Este nombre sirve como identificador único para la aplicación.

- Describe los componentes de la aplicación; actividades, servicios, *broacast receivers* y cualquier *content provider* que contenga la aplicación. Le da nombre a las clases que implementan cada uno de los componentes y publica sus capacidades (por ejemplo, qué mensajes pueden manejar). Estas declaraciones permiten conocer al sistema Android cuáles son los componentes y bajo qué condiciones pueden ser lanzados.
- Determina qué procesos albergarán los componentes de la aplicación.
- Declara qué permisos debe tener la aplicación de modo que pueda acceder a partes protegidas del API e interactuar con otras aplicaciones.
- También declara qué permisos deben tener otras aplicaciones para que éstas puedan interactuar con los componentes de la aplicación.
- Lista las clases de instrumentación que proporcionan perfiles y otra información mientras la aplicación se está ejecutando. Estas declaraciones sólo tienen sentido cuando la aplicación está siendo desarrollada y probada. Finalmente son borradas antes de que la aplicación sea publicada.
- Declara el mínimo nivel de API que requiere la aplicación de Android.
- Lista las librerías contra las que la aplicación debe ser enlazada.

Se puede obtener más información acerca del fichero de manifiesto y sus diversas opciones en [23].

2.1.6. Activity

Una Activity es una cosa única y concreta que un usuario puede hacer [24]. Casi todas las actividades interactúan con el usuario, así que la clase Activity se ocupa de la creación de la ventana en la que el usuario puede situar la interfaz gráfica mediante el método `setContentView(View)`. Aunque las actividades frecuentemente son mostradas al usuario como ventanas a tamaño completo, también pueden ser utilizadas de distintos modos: como una ventana flotante o también metida dentro de otra Activity. Hay dos métodos que casi todas las sub-clases de una Activity implementarán:

- `onCreate(Bundle)`: donde se inicializa la Activiy. Más importante, aquí será donde normalmente se llamará al método `setContentView(int)` con el recurso que define la interfaz de usuario (layout), y se usará el método `findViewById(int)` para obtener los elementos de la interfaz con los que se quiera interactuar.

- `onPause()`: donde se trata el hecho de que un usuario abandone la Activity. Más importante, cualquier cambio efectuado por el usuario debería ser confirmado (*commited*) en este punto. De no ser así, si se ha realizado algún cambio de estado (por ejemplo, se ha marcado una canción como favorita) y el sistema necesita recursos para realizar otras tareas, podría "matar" esta Activity y con ello se perdería el cambio realizado por el usuario.

Para ser de utilidad con el método `Context.startActivity()`, todas las clases Activity deben tener la correspondiente declaración en el AndroidManifest.xml mediante la etiqueta <`activity`>

Ciclo de vida de Activity

Las actividades en el sistema son gestionadas como si de una pila se tratara. Cuando una nueva Activity es iniciada, ésta es situada en lo más alto de la pila y pasa a ser la actividad en ejecución (las actividades previas siempre permanecen debajo de ella en la pila y no pasarán a un primer plano hasta que la nueva actividad concluya). Una actividad tiene esencialmente cuatro estados [25]:

- **Active o Running**: cuando una actividad se encuentra en primer plano de la pantalla. (en lo más alto de la pila).
- **Paussed**: si una actividad ha perdido el foco, pero aún así es visible (esto es, una nueva actividad transparente, o que no ocupe toda la pantalla ha sido iniciada) entonces se encuentra en este estado. Una actividad pausada está completamente viva (mantiene los datos y atributos y está ligada al WindowManager), pero puede ser matada por el sistema en situaciones de extrema escasez de memoria.
- **Stopped**: si una Activity está completamente tapada por otra, entonces se encuentra parada. Todavía mantiene todos los datos y atributos, sin embargo, ya no es visible para el usuario y será matada frecuentemente por el sistema si se necesita memoria en cualquier otro sitio.
- Si una actividad está pausada o parada, el sistema puede quitar la actividad de la memoria bien pidiéndole que termine o simplemente su proceso. Cuando se vuelve a mostrar al usuario, debe ser completamente restaurada a su estado previo.

La figura 2.2 muestra los estados y transiciones de una actividad. Los rectángulos representan métodos que se pueden implementar para realizar operaciones cuando una actividad se mueve de un estado a otro.

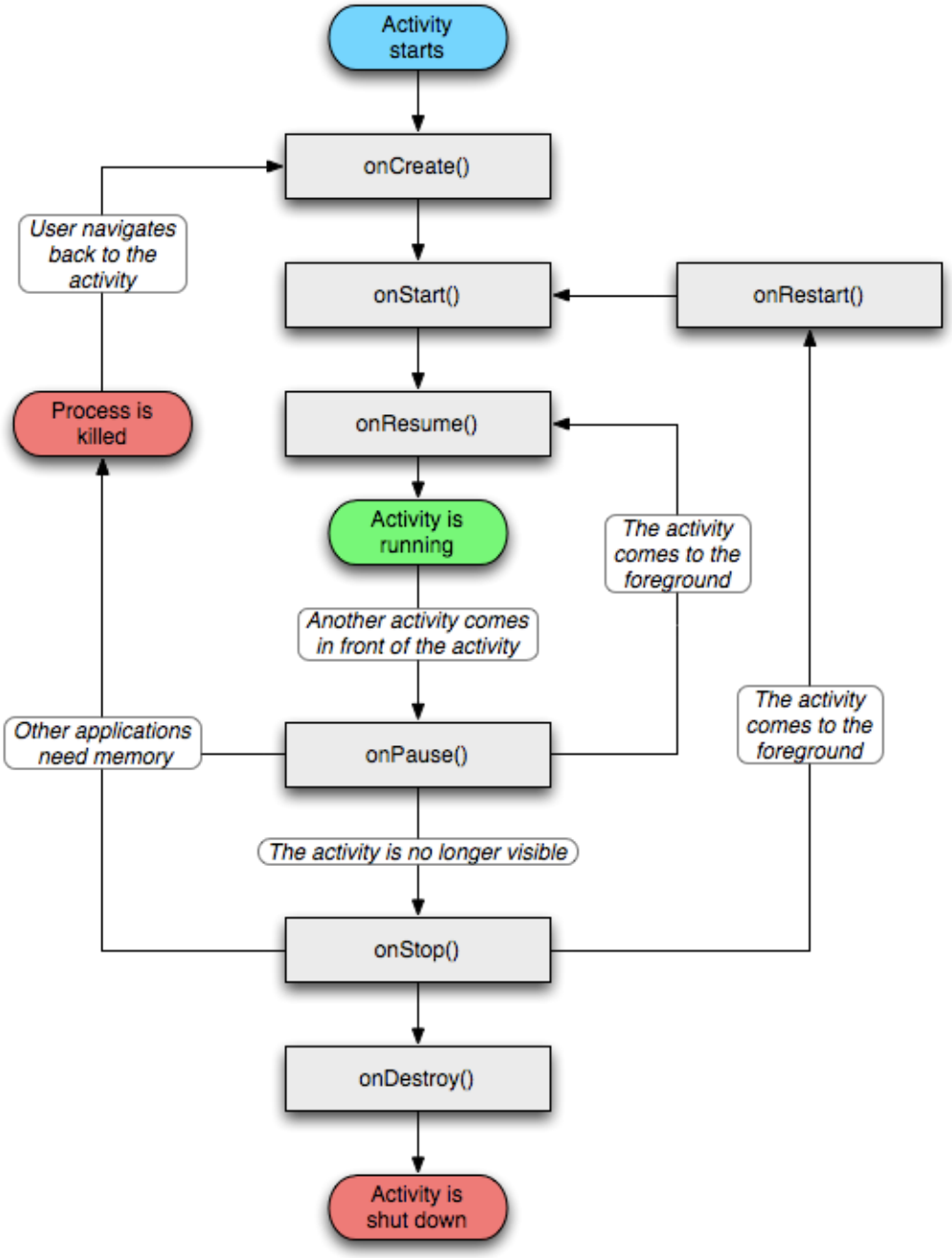

Figura 2.2: Ciclo de vida de un Activity.

Fuente: Activity de Android, Google Inc.

Hay tres ciclos clave que puede ser muy interesante monitorizar dentro de una actividad:

- El **ciclo de vida completo** de una actividad ocurre entre la primera llamada al método `onCreate(Bundle)` y la única (y última) llamada al método `onDestroy()`. Una actividad hará todas las labores genéricas de configuración de su estado en `onCreate()` y liberará todos los recursos pendientes en `onDestroy()`. Por ejemplo, si tiene un hilo ejecutándose en segundo plano para descargar datos de la red, se podría crear ese hilo en `onCreate()` y pararlo en `onDestroy()`.

- El **tiempo de vida VISIBLE** de una actividad ocurre desde la llamada al método `onStart()` hasta la correspondiente llamada a `onStop()`. Durante este tiempo el usuario puede ver la actividad en la pantalla, aunque puede que no esté en primer plano e interactuando con el usuario. Entre estos dos métodos se pueden mantener recursos necesarios para mostrar la actividad en pantalla. Por ejemplo, se puede registrar un `BroadcastReceiver` en el método `onStart()` para monitorizar cambios que afecten a la interfaz de usuario y eliminar el registro en el método `onStop()`, cuando el usuario ya no ve la actividad en la pantalla. Los métodos `onStart()` y `onStop()` pueden ser llamados múltiples veces, tantas como la actividad se convierta en visible o invisible.

- El **tiempo de vida en el que una actividad está en primer plano** ocurre entre los métodos `onResume()` y la correspondiente llamada al método `onPause()`. Durante este tiempo la actividad está delante de todas las demás y está interactuando con el usuario. Una actividad puede ir frecuentemente entre los estados pausado y resumido así que el código en estos métodos debería ser bastante ligero.

2.1.7. Servicios

Un servicio no tiene una interfaz de usuario visual, sino que se ejecuta en un segundo plano por un periodo de tiempo indeterminado [26] [27]. Por ejemplo, un servicio podría reproducir archivos musicales mientras que el usuario realiza otras tareas. Cada servicio extiende a la clase Service

Nótese que los servicios, como otros objetos de la aplicación, se ejecutan en el hilo principal del proceso donde están corriendo. Esto significa que, si el servicio va a hacer uso intensivo de la CPU (reproducción de mp3), debería lanzar su propio `Thread` en el cuál ejecutar dicha funcionalidad.

¿Qué es un Servicio?

La mayor parte de la confusión acerca de los servicios en realidad gira en torno a lo que no es:

- Un servicio no es un proceso separado. El objeto `Service` en sí mismo no implica que se ejecute en su propio proceso. A menos que sea especificado de otra manera, el servicio se ejecuta en el mismo proceso en el que se está ejecutando la aplicación.
- Un servicio no es un hilo. No es una forma de hacer trabajo fuera del hilo principal de la aplicación (para evitar que la aplicación deje de responder).

Así, un servicio es muy simple y provee dos características principales:

- Una facilidad para la aplicación para notificar al sistema que quiere hacer algo en segundo plano (incluso cuando el usuario no está interactuando directamente con la aplicación). Esto se corresponde con la llamada a `Context.startService()`, que pide al sistema que programe un trabajo para un servicio y que sea ejecutado hasta que el servicio pare o hasta que alguien explícitamente lo pare.
- Una facilidad para una aplicación para que ésta exponga parte de su funcionalidad a otras aplicaciones. Esto se corresponde a las llamadas `Context.bindService()`, que permiten establecer una conexión de larga duración con un servicio para de este modo poder interactuar con él.

Cuando se crea un servicio por alguna de las razones expuestas, todo lo que en realidad hace el sistema es instanciar el componente y llamar a su método `onCreate()` y demás llamadas apropiadas en el hilo principal. Depende del servicio implementar esto con el comportamiento apropiado, como crear un hilo secundario en el que hacer el trabajo.

Nótese que debido a que el servicio en sí mismo es tan simple, se puede hacer la interacción con el servicio tan simple o complicada como uno quiera: desde tratarlo como un objeto java local sobre el cual hacer llamadas a sus métodos, hasta proporcionar una completa interfaz remota usando **AIDL**.

Ciclo de Vida de un Servicio

Hay dos motivos por los que un servicio puede ser ejecutado por el sistema [26]. Si alguien llama al método `Context.startService()` entonces el sistema obtendrá el servicio (creándolo y llamando a su método `onCreate()`

si es necesario) y después llamará al método `onStartCommand(Intent, int, int)`. En este punto el servicio se continuará ejecutando hasta que se llame al método `Context.stopService()` o `stopSelf()`. No importa cuantas veces se haya iniciado el servicio, una sola llamada a `stopService()` bastará para detenerlo. Sin embargo, los servicios pueden usar el método `stopSelf(int)` para asegurarse de que el servicio no es detenido hasta que los *intents* iniciados hayan sido procesados. En la figura 2.3 se aprecia el ciclo de vida de un servicio con más detalle.

Los clientes también pueden usar el método `Context.bindService()` para obtener una conexión persistente con el servicio. Esto también crea el servicio si no se está ya ejecutando (llamando al método `onCreate()`), pero no llama al método `onStartCommand()`. El cliente recibirá el objeto `IBinder` que el servicio devuelve en el método `onBind(Intent)`, permitiendo entonces al cliente hacer llamadas al servicio. El servicio permanecerá ejecutándose tanto tiempo como la conexión permanezca establecida (no importa si el cliente mantiene una referencia al objeto `IBinder`). Normalmente el `IBinder` devuelto por el servicio es un complejo interfaz que ha sido escrito usando **AIDL**.

Un servicio puede ser iniciado y también tener conexiones ligadas a él. En tal caso, el sistema mantendrá el servicio en ejecución tanto tiempo como si el servicio hubiese sido iniciado con `startService()` o hay una o más conexiones con él con el flag `Context.BIND_AUTO_CREATE`. Una vez que no se da ninguna de estas situaciones, el método `onDestroy()` del servicio es llamado para que éste sea terminado. Toda la labor de limpieza (parada de hilos, anular el registro de los *Receivers*) debería ser completa al regreso del método `onDestroy()`.

Permisos

El acceso global a un servicio se puede activar declarando la etiqueta <`service`> en el `AndroidManifest.xml`. Haciendo esto, otras aplicaciones necesitarán declarar el correspondiente <`uses-permission`> es su propio fichero de configuración para ser capaces de empezar, detener o crear conexiones contra el servicio. Además, un servicio puede protegerse de llamadas IPC con permisos, llamando al método *checkCallingPermision(String)* antes de ejecutar la implementación de la llamada.

Ciclo de Vida del Proceso

El sistema Android intentará mantener el proceso en el que se está ejecutando un servicio siempre que el servicio haya sido iniciado (método `start()`)

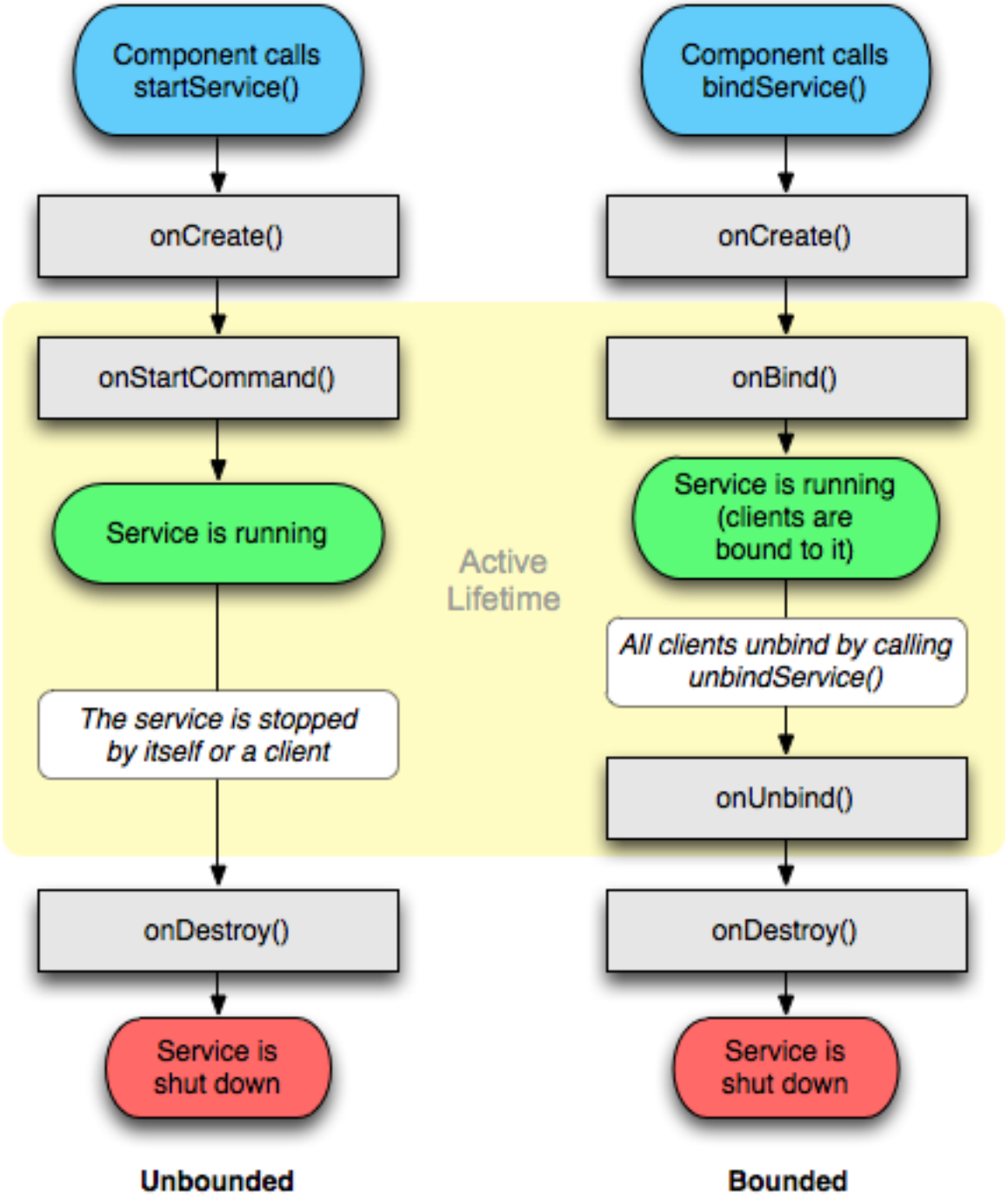

Figura 2.3: Ciclo de vida de un Servicio.

Fuente: Service de Android, Google Inc.

o tenga conexiones ligadas a él. Cuando se disponga de poca memoria y se necesite matar algún proceso existente, la prioridad de un proceso alojando un servicio será la más alta de las siguientes posibilidades [28]:

- Si el servicio está ejecutando código en ese momento en sus métodos `onCreate()`, `onStartCommand()` o `onDestroy()`, entonces el proceso que aloja el servicio será un proceso en primer plano para asegurarse de que ese código sea ejecutado sin que el proceso sea matado.
- Si el servicio ha sido iniciado, entonces el proceso que aloja el servicio es considerado menos importante que cualquier otro proceso que sea actualmente visible en la pantalla del usuario, pero más importante que cualquier otro proceso que no sea visible. Considerando que solamente unos pocos procesos son generalmente visibles al usuario, esto significa que el servicio no debería ser matado excepto en condiciones de extrema escasez de memoria.
- Si hay clientes ligados al servicio, entonces el proceso en el cuál se ejecuta el servicio nunca es menos importante que el cliente más importante. Esto es, si uno de los clientes es visible al usuario, entonces el propio servicio se considera que es visible también.
- Un servicio iniciado puede usar el método `startForeground(int, Notification)` para poner el servicio en un segundo plano, donde el sistema considera que es algo de lo que el usuario está activamente pendiente y que no es un candidato para ser matado en casos de baja memoria.

Nótese que esto significa que la mayor parte del tiempo que un servicio se está ejecutando, podría ser matado por el sistema bajo circunstancias de grandes dificultades de memoria. Si esto pasa, el sistema tratará más tarde de reiniciar el servicio. Una consecuencia importante de esto es que si se implementa el método `onStartCommand()` para planificar un trabajo que sea hecho de forma asíncrona o en otro hilo, entonces se podría querer usar el flag `START_FLAG_REDELIVERY` para que el sistema vuelva a pasar el mismo `Intent` en caso de reinicio de un servicio.

Modos de crear un Servicio Consolidado

Cuando se crea un servicio que proporciona vinculación a otros elementos, se debe proveer un objeto `IBinder` que ofrece la interfaz que los clientes pueden usar para interactuar con el servicio. Existen tres maneras de definir esta interfaz:

- Extendiendo la clase `IBinder`: si el servicio es privado a la aplicación y se ejecuta en el mismo proceso que el cliente, se debería crear una interfaz extendiendo la clase `Binder` y devolviendo una instancia de ella en el método `onBind()` [29].

- Utilizando la clase `Messenger`: si se necesita que la interfaz trabaje a través de diferentes procesos, se puede crear una interfaz para el servicio con la clase `Messenger`. [30].

- Utilizando AIDL: AIDL realiza todo el trabajo de descomponer objetos en primitivas que el sistema operativo puede entender y así enviarlas de unos procesos a otros (IPC).

AIDL

AIDL, Android Interface Definition Languaje en sus siglas en inglés [31], permite definir una interfaz que ambos cliente y servicio utilizarán para comunicarse entre ellos utilizando lo que se denomina *Comunicación entre Procesos (IPC)*. En Android, un proceso normalmente no puede acceder a la memoria de otro proceso. Así, para comunicarse necesitan descomponer sus objetos en primitivas que el sistema operativo puede comprender y así poder enviarlas de un sitio a otro. El código para hacer este proceso es muy tedioso, así que Android proporciona AIDL al programador para facilitar su tarea.

Para crear un servicio utilizando AIDL se necesitan realizar los siguientes pasos:

- **Crear el fichero .aidl**: este archivo define la interfaz de programación con declaraciones de métodos. AIDL utiliza JAVA y una sintaxis simple que permite declarar la interfaz con uno o más métodos que pueden llevar parámetros y tener valores de retorno. Los parámetros y retornos pueden ser de cualquier tipo, incluso otras interfaces AIDL.

- **Implementar la interfaz**: el SDK de Android genera una interfaz JAVA basado en el fichero .aidl. Esta interfaz tiene una clase interna y abstracta llamada `Stub`, que extiende a `Binder` e implementa los métodos de la interfaz AIDL. El programador debe extender la clase `Stub` e implementar los métodos.

- **Exponer la interfaz a los clientes**: implementar la clase `Service` y sobreescibir el método `onBind()` para que devuelva la clase `Stub` implementada.

2.1.8. Broadcast Receivers

Un `BroadcastReceiver` es un componente que no hace nada excepto recibir y reaccionar ante anuncios [32]. Muchos de estos anuncios los origina el propio sistema (por ejemplo, anuncios de que ha cambiado la zona horaria, de que la batería está baja o de que el usuario ha cambiado el idioma del sistema). Las aplicaciones también pueden crear anuncios como por ejemplo hacer saber a otras aplicaciones que se ha descargado algo o que el dispositivo está listo para usarse.

Una aplicación puede tener el número que quiera de componentes que respondan a los anuncios que se consideren importantes. Todos extenderán la clase `BroacastReceiver`. `BroadcastReceivers` no muestran una interfaz de usuario. Sin embargo, sí que pueden iniciar cualquier actividad como respuesta a un anuncio que hayan captado o pueden notificárselo al usuario a través de `NotificationManager`. Las notificaciones pueden captar la atención del usuario de diversas maneras: iluminando el dispositivo, haciéndolo vibrar, mediante sonidos, etc. Normalmente, mostrarán un icono en la barra de estado del dispositivo que el usuario podrá abrir para ver el mensaje de notificación.

Los `BroadcastReceivers` se pueden registrar dinámicamente dentro de la aplicación o declarando el registro con la etiqueta <receiver> en el fichero *AndroidManifest.xml*.

NOTA: si se registra un `BroadcastReceiver` en el método `onResume()` de una actividad, éste debería borrarse en el método `onPause()`, de este modo no se recibirán notificaciones cuando la actividad está pausada, evitando sobrecargas innecesarias. No se debe eliminar el registro de un `BroadcastReceiver` en el método `onSaveInstanceState()`, porque este método no será llamado si el usuario vuelve atrás en el historial.

Hay dos clases principales de emisiones (`broadcast`) que se pueden recibir:

- **Broadcasts Normales**: enviados con `Context.sendBroadcast()`, son completamente asíncronos. Todos los receptores del *broadcast* se ejecutan en un orden sin definir, con frecuencia al mismo tiempo. Esto es más eficiente, pero significa que los receptores no pueden usar el resultado.

- **Broadcasts Ordenados**: enviados con `Context.sendOrderedBroadcast()`, se entregan a un receptor cada vez. A medida que cada receptor se ejecuta, éste puede propagar un resultado al siguiente receptor o puede abortar el *broadcast* por completo de modo que no será propagado a ningún otro receptor. El

orden en que se ejecuta cada receptor puede ser configurado con el atributo `android:priority`.

Incluso en el caso de los *broadcast* normales, en ciertas ocasiones el sistema puede decidir entregar el *broadcast* a los receptores de uno en uno. Particularmente, si los receptores requieren la creación de un proceso, solamente se ejecutará uno cada vez para evitar sobrecarga en el sistema con los nuevos procesos. En esta situación, sin embargo, se mantienen las restricciones de los *broadcast* normales: estos receptores no podrán devolver resultados ni abortar el *broadcast*.

Nótese que, aunque se está usando la clase `Intent` para el envío y recepción de *broadcasts*, este mecanismo es completamente distinto al usado para iniciar actividades con el método `Context.startActivity()`. No hay forma de que un `BroadcastReceiver` pueda ver o capturar *Intents* usados con `startActivity`. Igualmente, cuando se hace una difusión de un *Intent*, nunca se encontrará un iniciará una actividad. Estas dos operaciones son semánticamente muy distintas: iniciar una Actividad con un *Intent* es una operación en primer plano que modifica aquello con lo que el usuario está interactuando, mientras que difundir un *Intent* es una operación en segundo plano de la que el usuario normalmente no tiene constancia.

Ciclo de vida de un Receptor

Un objeto `BroadcastReceiver` sólamente es válido mientras dure la llamada al método `onReceive(Context, Intent)`. Una vez que el código vuelve de esta función, el sistema considera que el objeto ya está terminado e inactivo. Esto tiene importantes repercusiones sobre lo que se puede hacer en la implementación del método `onReceive(Context, Intent)`: cualquier cosa que requiera operaciones asíncronas no está disponible, dado que se necesitaría volver de la función para manejar la operación asíncrona, pero en ese punto el `BroadcastReceiver` ya no está activo por lo que el sistema sería libre de matar su proceso antes de que la operación asíncrona terminase.

En particular, no se puede mostrar un diálogo o enlazarse a un servicio desde dentro de un `BroadcastReceiver`. Para los primeros, se debería utilizar el API de `NotificationManager` en su lugar. Para los últimos, se puede utilizar `Context.startService()` para mandar un comando a un servicio.

Permisos

Los permisos de acceso pueden ser aplicados por el receptor o por el remitente de un `Intent`.

Para aplicar un permiso en el envío, se pasará como argumento un permiso no nulo a los métodos `sendBroadcast(Intent, String)` o `sendOrderedBroadcast(Intent, String, BroadcastReceiver, android.os.Handler, int, String, Bundle)`. Sólo los receptores a los que se les haya concedido este permiso (usando la etiqueta `<uses-permission>` en el fichero *AndroidManifest.xml*) serán capaces de recibir el *broadcast*.

Para aplicar un permiso en la recepción, se utiliza un permiso no nulo cuando se registra el receptor –bien llamando al método `registerReceiver(BroacastReceiver, IntentFilter, String, android.os.Handler)` o bien en el *AndroidManifest.xml* por medio de la etiqueta `<receiver>`. Sólo se permitirá enviar *broadcasts* a aquellos elementos que tengan el permiso adecuado.

Ciclo de vida del proceso

Un proceso que está ejecutando actualmente un `BroadcastReceiver` (esto es, ejecutando código en su método `onReceive(Context, Intent)`), es considerado un proceso en primer plano, y seguirá siendo ejecutado por el sistema excepto en casos extremos de escasez de memoria.

Una vez que se vuelve de `onReceive()`, el `BroadcastReceiver` no está activo y el proceso donde se ejecuta es igual de importante como lo sea cualquier otro componente de la aplicación que se esté ejecutando en él. Esto es especialmente importante porque si en el proceso solamente se estaba ejecutando el `BroadcastReceiver`, entonces a la vuelta del `onReceive()` el sistema considerará que el proceso está vacío y por tanto lo matará para liberar recursos que puedan ser utilizados por procesos más importantes.

Esto significa que para operaciones de larga duración se usará frecuentemente un servicio junto con un `BroadcastReceiver` para mantener el proceso contenedor activo durante todo el periodo que dure la operación.

2.1.9. Content Providers

Un `ContentProvider` hace que un conjunto específico de datos de la aplicación esté disponible para otras aplicaciones [33]. Los datos pueden ser almacenados en el sistema de ficheros, en una base de datos SQLite o de cualquier otra manera que tenga sentido. El Content Provider extiende la clase `ContentProvider` e implementa una serie de métodos estándar que permita a las aplicaciones acceder a los datos y almacenarlos. Sin embargo, las aplicaciones no llaman a estos métodos directamente. Más bien, usan

un objeto `ContentResolver`. Estos objetos pueden "hablar" con cualquier `ContentProvider`.

2.1.10. Intents

Un `Intent` es una descripción abstracta de una operación que se va a realizar [34]. Puede utilizarse para lanzar actividades, enviar *broadcasts* a `BroadcastReceivers`, iniciar servicios o comunicarse con servicios en ejecución.

Un `Intent` proporciona facilidades para enlazar código en diferentes aplicaciones en tiempo de ejecución. Su uso más significativo es para lanzar actividades, donde se puede considerar como el pegamento entre actividades. Básicamente, se trata de una estructura de datos pasiva que contiene una descripción abstracta de la acción que se quiere realizar. Las principales piezas de información de un `Intent` son:

- **action**: la acción general que se va a realizar, tales como `ACTION_VIEW`, `ACTION_EDIT`, `ACTION_MAIN`, etc.
- **data**: los datos sobre los que operar, tales como contatos, personas, expresados como un Uri.

Además de estos atributos primarios, existen otros secundarios que se pueden incluir con un `Intent`:

- **category**: da información adicional acerca de la acción a ejecutar.
- **type**: especifica el tipo de datos del Intent. Normalmente el tipo se infiere de los datos mismos. Configurando este atributo, se desactiva la evaluación y se fuerza un tipo explícito.
- **component**: especifica el nombre explícito de la clase de componente que se usa para el Intent. Normalmente esta información se determina mirando el resto de información del Intent. Si este atributo está configurado entonces toda la evaluación previa se anula. Especificando este atributo convierte al resto en opcionales
- **extras**: esto es un objeto `Bundle` con cualquier tipo de información adicional. Se puede utilizar para proporcionar información extendida al componente.

Activando componentes

Hay diferentes métodos para activar cada tipo de componente [35]:

- Una actividad puede ser lanzada llamando a los métodos `Context.startActivity()` o `Activity.startActivityForResult()`, pasándole a cada uno de ellos un objeto `Intent`. La actividad lanzada puede obtener la información acerca del `Intent` inicial llamando al método `getIntent()`.

- Con frecuencia, una actividad lanza la siguiente. Si la primera espera un resultado de la segunda, llamará al método `startActivityForResult()` en vez de `startActivity()`. Por ejemplo, si se lanza una actividad para que el usuario seleccione una foto, se podría esperar que devuelva la foto que ha sido elegida. El resultado es devuelto en un objeto `Intent` pasado al método `onActivityResult()` de la actividad que inició la llamada.

- Un servicio es iniciado mediante la llamada al método `Context.startService()`, pasándole un objeto `Intent`. Android llamará al método `onStart()` del servicio y le pasará dicho objeto `Intent`. De forma similar, se puede llamar al método `Context.bindService()`, pasándole el objeto `Intent`, para establecer una conexión entre el componente de llamada y un servicio destino. El servicio recibe el objeto `Intent` en una llamada al método `onBind()`. Por ejemplo, una Activity podría establecer una conexión con el servicio de reproducción de música para ofrecer al usuario una interfaz con la que manejar tal reproductor. La Activity llamaría al método *bindService* para establecer esa conexión y luego llamar a los métodos definidos por el servicio que puedan afectar a la reproducción

- Un aplicación puede iniciar una emisión pasando un objeto `Intent` a métodos como `Context.sendBroadcast()`, `Context.sendOrderedBroadcast()`, y `Context.sendStickyBroadcast()`, en cualquiera de sus variantes. Android entrega el `Intent` a todos los recibidores de emisiones interesados llamando a sus métodos onReceive.

Resolución de Intents

Hay dos formas principales de `Intents` que un usuario utilizará [36]:

- Los **Intents explícitos** han especificado un componente (usando el método `setComponent(componentName)` o `setClass(Context,`

`Class)`), lo que proporciona la clase exacta a ser ejecutada. Con frecuencia esto no incluirá ninguna otra información, siendo simplemente una forma de lanzar Activities por parte de la aplicación.

- Los **Intents implícitos** no han especificado ningún componente. En vez de eso, deben incluir suficiente información para que el sistema determine cual de los componentes disponibles es el mejor para ejecutar la acción descrita en el `Intent`.

Cuando se usan Intents implícitos, se tiene que saber qué hacer con ellos. Esto es manejado por el proceso *Intent Resolution* (Resolución de Intents), el cuál mapea un Intent a una Activity, BroadcastReceiver o servicio.

El mecanismo de Resolución de Intents básicamente lo que hace es comparar un Intent con todas las líneas <`intent-filter`> en los paquetes de la aplicación instalados (además, en caso de broadcasts, busca en todos los objetos registrados con el método `registerReceiver(BroadcastReceiver, IntentFilter)`).

Cerrando componentes

Un Content Provider está activo únicamente cuando está respondiendo a una petición proviniente de un objeto `ContentResolver`. Y un BroadcastReceiver está activo solo cuando está respondiendo a un mensaje broadcast. Así que no hay una necesidad de cerrar esos componentes explícitamente. Por otra parte, las actividades proporcionan la interfaz de usuario. Están en una larga conversación con el usuario y pueden permanecer activas, incluso si no están siendo utilizadas, mientras la conversación continua. De forma similar, los servicios pueden permanecer ejecutándose un largo periodo de tiempo. Así que Android tiene métodos para cerrar actividades y servicios de forma ordenada:

- Una Activity puede ser cerrada llamando a su método `finish()`. Una Activity puede cerrar otra Activity llamando al método `finishActivity()`.

- Un servicio puede ser parado llamando a su método `stopSelf()` o llamando al método `Context.stopService()`.

Los componentes también podrían ser cerrados por el sistema cuando éstos no vayan a ser usados más o cuando Android reclame memoria para otros componentes activos.

2.1.11. Interfaz de Usuario

En una aplicación Android, la interfaz de usuario se construye utilizando los objetos `View` y `ViewGroup` [37]. Hay muchos tipos de vistas y grupos de vistas, cada uno de los cuales son descendientes de la clase `View`. Los objetos `View` son las unidades básicas de la interfaz de usuario en la plataforma Android. Sirven como base para las subclases llamadas `Widgets`, que proporcionan elementos de la interfaz de usuario totalmente implementados, tales como botones, campos de texto, etc. La clase `ViewGroup` sirve como la base para las subclases llamadas "layouts", que ofrecen distintos tipos de diseño para la interfaz, tales como linear, relativo, tabular.

Un objeto `View` es una estructura de datos cuyas propiedades almacenan los parámetros del diseño y el contenido para un área rectángular específica de la pantalla. El objeto `View` maneja sus propias medidas, diseño, cambio de foco, scroll, e interacción con el usuario por medio de gestos o teclas para el área rectangular de la pantalla en la que reside. Como objeto de la interfaz de usuario, también es punto de interacción con el usuario.

Jerarquía de Vistas

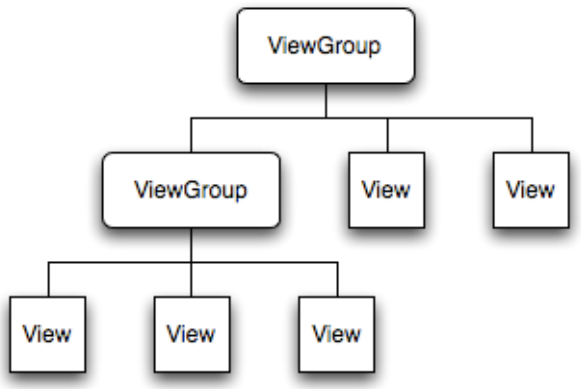

Figura 2.4: Jerarquía de vistas.

Fuente: Google Inc.

En la plataforma Android, se define la interfaz de usuario utilizando una jerarquía de nodos de vistas y grupos de vistas, tal y como se muestra en la figura 2.4. Este árbol jerárquico puede ser tan simple o complejo como se necesite y se puede construir usando los elementos predefinidos por la plataforma Android, o bien creando las vistas uno mismo.

Para fijar el árbol jerárquico de vistas a la pantalla para su presentación, la Activity debe llamar al método `setContentView()`, pasándole la referencia al objeto raíz del árbol.Android recibe esta referencia y la usa para invalidar, medir y dibujar el árbol. El nodo raíz de la jerarquía solicita que sus hijos se dibujen ellos mismos (cada `ViewGroup` es responsable de llamar a cada una de sus vistas hijas para que se dibujen ellas mismas). Los hijos pueden solicitar un determinado tamaño o localización dentro del padre, pero es éste último el que tiene la decisión final sobre cómo y dónde puede estar cada hijo. Android *parsea* los elementos de la interfaz en orden (desde lo más alto del árbol jerárquico), instanciando las vistas y añadiéndolas a sus padres. Debido a que son pintadas en orden, si hay elementos que comparten una misma posición, la última que sea pintada será la que permanecerá encima.

Layout

La manera más común de definir el diseño de interfaz de usuario y mostrar la jerarquía de vistas es mediante un fichero XML [38], de modo que sea más fácilmente entendible su lectura por personas. Cada elemento en el XML es o bien un objeto `View` o uno `ViewGroup` (o un descendiente de ellos). Los objetos `View` representan hojas del árbol, mientras que los `ViewGroup` son las ramas. El nombre de un elemento XML es el respectivo a la clase JAVA que representa. De este modo, un elemento <`TextView`> crea un objeto de tipo `TextView` en la interfaz de Usuario, y un elemento <`LinearLayout`> crea un `ViewGroup` de tipo `LinearLayout`. Cuando se carga el fichero XML, Android inicializa estos objetos en tiempo de ejecución que se corresponden con los elementos declarados en el diseño.

Widgets

Un Widget es un objeto `View` que sirve como interfaz para la interacción del usuario con la aplicación [39]. Android proporciona una serie de Widgets ya implementados, como botones, checkboxes, campos de texto, etc. que se puede utilizar para construir una interfaz de usuario rápidamente. Algunos de los Widgets que proporciona Android son más complejos, como el selector de fechas, un reloj y controles de *zoom*. Pero el usuario no está limitado a los tipos de Widgets proporcionados por Android. Si se quiere algo más personalizado, se pueden crear Widgets propios definiendo un objeto `View` propio o combinando Widgets existentes.

Menús

Los menús de la aplicación son otra parte importante de la interfaz gráfica [40]. Ofrecen una interfaz fiable que revela funciones y configuraciones de la aplicación. El menú de aplicación más común se muestra pulsando el botón MENU en el dispositivo. Sin embargo, se pueden añadir también menús contextuales que serán mostrados cuando el usuario mantenga pulsado un *item* en la pantalla. Los menús también están estructurados utilizando una jerarquía de vistas, pero esta estructura no la define el desarrollador de la aplicación. En lugar de eso, el desarrollador define los métodos `onCreateOptionsMenu()` y `onCreateContextMenu()` para la Activity y declara los items que se quieren incluir en el menú. Cuando sea apropiado, Android creará automáticamente la jerarquía de vistas necesaria para el menú, y dibujará los items en ella.

Los menús también manejan sus propios eventos, así que no hay necesidad de registrar los *listeners* para los items que componen el menú. Cuando un item es seleccionado, el método `onOptionsItemSelected()` o `onContextItemSelected()` será llamado por el sistema.

También, tal y como en el resto de vistas, los elementos del menú se pueden definir a través de un fichero XML.

Eventos del Interfaz de Usuario

Una vez que se han añadido *Views/Widgets* al interfaz de usuario, probablemente el siguiente paso será conocer la interacción del usuario con ellos [41]. Para esto, se puede hacer una de estas dos cosas:

- Definir un *event listener*, y registrarlo con la vista. La mayoría de las veces, será de esta forma como se escucharán los eventos de la interfaz de usuario. La clase `View` contiene una colección de interfaces anidados llamados `On_<`*acción*`>_Listener`, cada una con un método de devolución de llamada llamado `On_<`*acción*`>()`. Por ejemplo, `View.OnClickListener` (para manejar "clicks" en una vista), `View.OnTouchListener()` (para manejar eventos en la pantalla táctil). Así que, si se quiere que una vista sea notificada si se hace click en ella, habría que implementar `OnClickListener` y definir su método `onClick()` y registrarlo a la vista usando para ello el método `setOnclickListener()`.

- Sobreescribir un método de devolución de llamada ya existente para la clase `View`. Esto es lo que se debería hacer cuando se ha implementado

una clase `View` propia y se quieren escuchar eventos específicos ocurridos dentro de ella. Esto permite definir el comportamiento por defecto de cada evento dentro de la vista propia.

2.1.12. Almacenamiento de Datos

Android ofrece varias posibilidades para almacenar de forma persistente los datos de una aplicación [42]. La solución que el desarrollador escoja dependerá de las necesidades específicas que se tengan, tales como si los datos de la aplicación serán compartidos o privados, o cuánto espacio se necesitará para almacenarlos.

Las opciones para almacenar estos datos son:

- **Shared Preferences** (Preferencias compartidas): almacenan datos privados de tipo primitivo en formato clave-valor.
- **Almacenamiento Interno**: guardan los datos privados en la memoria del dispositivo.
- **Almacenamiento Externo**: guardan datos públicos en el almacenamiento externo compartido.
- **Base de datos SQLite**: almacena datos estructurados en una base de datos privada.
- **Conexión de Red**. Almacena datos en la web con su propio servidor.

Además, Android proporciona al usuario una manera de exponer incluso los datos privados de una aplicación a otras aplicaciones, usando los **Content Provider**. Un Content Provider es un componente opcional que expone acceso de lectura y/o escritura a los datos de una aplicación, con las restricciones que el desarrollador quiera imponer.

Preferencias Compartidas

La clase `SharedPreferences` proporciona un entorno de trabajo genérico que permite guardar y acceder a datos en forma de pares de clave-valor [43]. Estos datos han de ser de cualquier tipo primitivo: `boolean`, `float`, `int`, `long`, y `string`. Estos datos persistirán a través de las sesiones de usuario (incluso si la aplicación es matada).

- `getSharedPreferences()`: este método se usa cuando hay múltiples ficheros de preferencias y se indica el nombre del mismo mediante el primer parámetro.

- `getPreferences()`: usado cuando solo hay un fichero de preferencias.

Para escribir datos:

1. Llamar al método `edit()` para obtener un objeto `SharedPreferences.Editor`.
2. Añadir valores con métodos como `putBoolean()`, `putString()`, etc.
3. Confirmar los nuevos valores con `commit()`.

Para leer valores:

1. Métodos como `getBoolean()`, `getString()`, etc.

Almacenamiento interno

Se pueden salvar archivos directamente en el almacenamiento interno del dispositivo [44]. Por defecto, los ficheros salvados en el almacenamiento interno son privados para la aplicación, por lo que el resto de aplicaciones no podrán acceder a ellos. Cuando el usuario desinstale la aplicación, estos datos se borrarán.

Para crear y escribir en un fichero en el almacenamiento interno:

1. Llamar al método `openFileOutput()`, con el nombre del fichero y el modo de operación. Esto devuelve un objeto `FileOutputStream`.
2. Escribir en el fichero con el método `write()`.
3. Cerrar el flujo de datos con `close()`.

Guardar ficheros cache Si se quieren guardar en memoria *cache* algunos datos, más que almacenarlos de forma persistente, se debería utilizar el método `getCacheDir()` para abrir un fichero que represente el directorio interno donde la aplicación debería guardar ficheros temporales cacheados. Cuando el dispositivo dispone de poco espacio de memoria interna, Android puede borrar estos ficheros temporales para recuperar algo de espacio. Sin embargo, no se debería confíar esta labor de limpieza al sistema, sino que debe ser el desarrollador el que maneje este espacio consumido por los ficheros cacheados y mantenerlo en una cifra razonable, tal como 1MB. Cuando el usuario desinstala la aplicación, estos datos son borrados.

Almacenamiento Externo

Todos los dispositivos que son compatibles con Android soportan almacenamiento externo que se puede utilizar para guardar ficheros [45]. Esto puede ser un dispositivo de almacenamiento externo (como una tarjeta SD) o un dispositivo interno. Los ficheros guardados en el almacenamiento externo son públicos y pueden ser modificados/leídos por un usuario.

Comprobar la disponibilidad del medio de almacenamiento Antes de que se puede hacer nada con el almacenamiento externo, siempre se debería llamar al método `getExternalStorageState()` para comprobar si el medio está disponible. El medio puede estar montando en una computadora, perdido, con estado de solo lectura o en algún otro estado.

Acceder a ficheros del almacenamiento externo Si se está utilizando un **API 8** o posterior, se usa el método `getExternalFilesDir()` para abrir un objeto `File` que representa un directorio de almacenamiento externo donde se deberían almacenar los ficheros. Este método toma como primer parámetro *type*, que especifica el tipo de subdirectorio que se quiere, tales como MÚSICA o TONOS DE LLAMADA. Especificando el tipo de directorio, es una forma de asegurarse de que Android categorice adecuadamente los ficheros multimedia en el sistema. Si el usuario desinstala la aplicación, el directorio y sus contenidos serán borrados.

Si se está utilizando un API nivel 7 o anterior, se usa el método `getExternalStorageDirectory()` para abrir el objeto `File` que representa la raíz del almacenamiento externo. Se deberían escribir los datos en el siguiente directorio:

/Android/data/-packagename-/files/

El parámetro -packagename- es el nombre del paquete de la aplicación (estilo JAVA). Si el dispositivo del usuario está ejecutando con un API 8 o superior, al desinstalar el usuario la aplicación también se borrarán el directorio y todo su contenido.

Base de Datos

Android provee completo soporte a bases de datos **SQLite** [46]. Cualquier base de datos que se cree, será accesible por nombre por cualquier clase de la aplicación, pero no fuera de la aplicación.

La forma recomendada para generar una base de datos SQLite nueva es crear una subclase de `SQLiteOpenHelper` y sobrescribir el método `onCreate()`, en el cuál se pueden ejecutar comandos SQLite para crear las tablas de la base de datos.

Se puede obtener una instancia de la implementación de

`SQLiteOpenHelper` usando el constructor que se haya definido. Para escribir y leer de la base de datos, se llama al método `getWritableDatabase()` y `getReadableDatabase()` respectivamente. Ambos métodos devuelven un objeto `SQLiteDatabase` que representa la base de datos y proporciona métodos para las operaciones SQLite.

Se pueden ejecutar consultas SQLite utilizando los métodos `SQLiteDatabase.query()`, que aceptan diversos parámetros de entrada tales como tabla, proyección, selección, columnas, grupos, etc. Para consultas complejas, como aquellas en las que se necesitan alias, se debería utilizar `SQLiteQueryBuilder`, que ofrece métodos convenientes para construir consultas. Cada consulta SQLite devolverá un objeto `Cursor` que apunta a todas las filas devueltas por la consulta. El `Cursor` es siempre el mecanismo por el que navegar por los resultados de una consulta y leer las filas y columnas devueltas.

Android no impone ninguna limitación más allá de los conceptos de SQLite estándar. Se recomienda incluir claves auto incrementales para que los registros puedan ser rápidamente encontrados. Esto no es obligatorio para datos privados, pero si se implementa un Content Provider, se debe incluir un identificador único utilizando la constante `BaseColumns._ID`

Conexión de Red

Se puede usar la red (cuando esté disponible) para almacenar y obtener datos de los servicios web. Para hacer operaciones de red, se usan las clases de los siguientes paquetes: `java.net.*` y `android.net.*` [47].

2.2. RSS y archivos multimedia

2.2.1. ¿Qué es RSS?

Se trata de un formato para compartir datos, definido en la versión 1.0 del estándar XML [48]. Uno puede entregar información en este formato y otros pueden recibir ésta y otras informaciones de diversas fuentes en el mismo formato. La información proporcionada por una página web en este formato es llamada *RSS Feed*. Los navegadores más recientes pueden leer directamente los ficheros RSS, pero igualmente se pueden utilizar herramientas especiales o agregadores para este fin [49]. En la figura 2.5 se muestra un esquema del proceso que se lleva a cabo para hacer uso de esta tecnología.

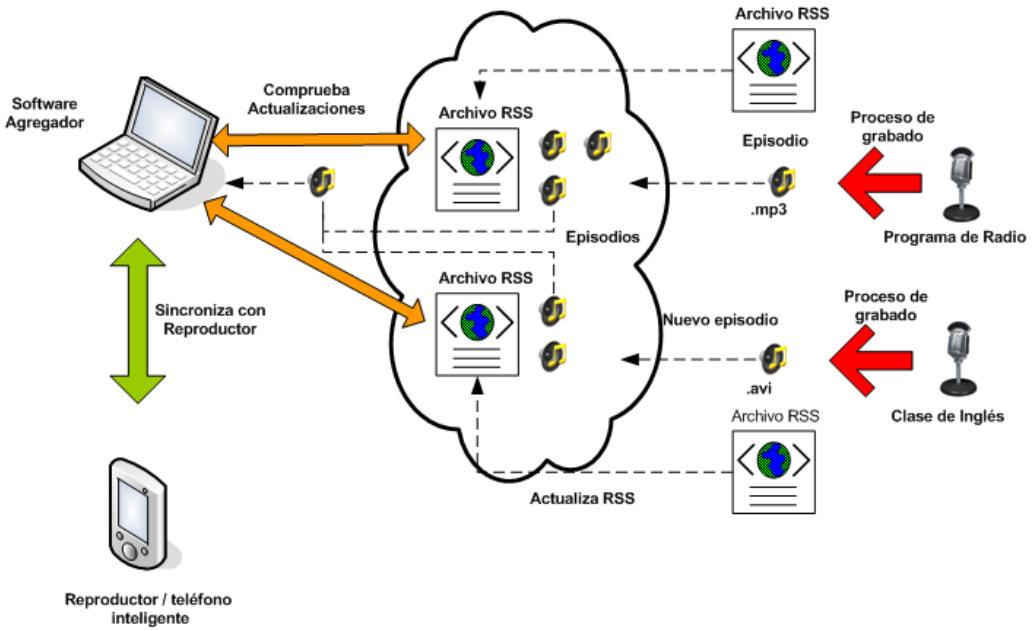

Figura 2.5: Proceso de publicación y uso de RSS.

2.2.2. ¿Por qué usar RSS?

Para obtener información o noticias proporcionadas por páginas web en un formato que los ordenadores personales puedan procesar. También para mostrar esta información en una página web o para ser leída por un usuario. Para el proveedor de la información, RSS permite mandar noticias.

2.2.3. ¿Cómo funciona RSS?

Publicar artículos y noticias en la web usando RSS es muy sencillo:

1. Primero se necesitan páginas web que vayan a ser mostradas en otros *websites*. Este conjunto de páginas serán los *feeds* RSS.

2. Se necesita un fichero XML que defina el feed RSS. Este fichero contiene la URL, el título y un resumen de cada página a ser mostrada.

3. Una persona que quiera leer el feed en su ordenador. Esta persona usará un lector de RSS o su navegador para obtener el feed con el comando apropiado.

4. Si es otra página web la que quiere mostrar el feed, tendrá que cargar el fichero RSS del proveedor, para de esta forma obtener las URLs, títulos y resúmenes de los feeds. Esto puede realizarse mediante un script PHP.

5. Cuando alguien visite la página web del receptor, el script PHP se ejecutará, y mostrará la información obtenida del fichero RSS del proveedor.

6. Haciendo "click" en una línea del listado, se mostrará la página del proveedor.

2.2.4. Estructura de un documento RSS

Es un fichero XML cuyo contenedor global es la etiqueta <RSS> (formato 2.0). El fichero contiene al menos un canal (<channel>), que es la página web que proporciona la información. El canal también proporciona algunos artículos o datos, que son páginas web del mismo u otros sitios [49].

Principales etiquetas RSS

- **rss**: el contenedor global.
- **channel**: el canal de distribución. Tiene muchas etiquetas descriptivas y contiene uno o muchos items.

Etiquetas requeridas para <channel>

A continuación se muestra un listado con los elementos obligatorios de <channel> [50]:

- **title**: el nombre del canal. Es cómo la gente se refiere al servicio. Si un sitio web HTML contiene la misma información que el fichero RSS, el título del canal debería ser el mismo que el título del sitio web.
- **link**: URL de la página web que provee este canal.
- **description**: descripción del canal.
- **item**: uno al menos, para el contenido.

Etiquetas opcionales para <channel>

- **language**: el idioma usado para el texto.
- **image**: especifica un archivo PNG, JPEG o GIF que puede ser mostrado con el canal.
- **managingEditor**: dirección de correo electrónico de la persona responsable del contenido editorial
- **webMaster**: dirección de correo electrónico de la persona responsable de problemas técnicos relacionados con el canal.
- **pubDate**: fecha de publicación del contenido del canal. Todos las fechas aparecidas utilizan las especificación RFC 822, con la excepción del año, que puede ser expresado con 2 ó 4 dígitos.
- **lastBuildDate**: fecha de la última vez en la que el contenido del canal cambió.
- Existen más elementos. Ver especificación [49].

Elemento Item

Un canal puede contener cualquier número de items [51]. Un item puede representar una historia. Si es así, su descripción es el resumen de la historia, y en <link> enlaza con la historia completa. Un item puede ser completo en sí mismo, siendo la descripción el texto completo y pudiéndose omitir la etiqueta link.

Elementos de item:

Todos los elementos de item son opcionales, sin embargo, al menos <title> o <description> debe aparecer.

- **title**: título del item.
- **link**: la URL del item.
- **description**: resumen del artículo.
- **author**: dirección de correo electrónico del autor del item.
- **enclosure**: describe un objeto multimedia ligado al item.
- **pubDate**: fecha de publicación.
- **guid**: una cadena de caracteres única que identifica el item.

- **category**: la categoría del artículo.
- **source**: el canal RSS del cual proviene el item.
- **comments**: URL de una página web con comentarios acerca del item.
- **category**: incluye al item en una o varias categorías.

2.2.5. ¿Cómo se usa RSS?

Usando RSS en un escritorio

Los feeds RSS pueden ser mostrados directamente en los navegadores recientes. Se muestran a partir del Internet Explorer 7 y Firefox 2. Un lector RSS (o agregador) también puede ser instalado en el sistema operativo. La forma de acceder a los contenidos dependerá del lector empleado.

Utilizando un feed en una página web

Un feed RSS se muestra como una lista de títulos y, opcionalmente, resúmenes. Un click en el título mostrará la noticia o artículo. Los títulos se actualizarán automáticamente. Un script en PHP (o cualquier otro lenguaje de programación) construye la lista cada vez que la página es mostrada cargando el fichero RSS y extrayendo de él la información necesaria.

2.2.6. ¿Cómo publicar noticias en un feed RSS?

Hay diversas maneras de generar feeds RSS.

- Utilizando la librería RSS de PHP: un script PHP construirá el fichero XML con títulos y descripciones de páginas a partir de información de otra página.
- Utilizando una herramienta especializada para extraer datos de una página.

2.2.7. ¿Cómo saben los navegadores que hay un feed RSS en una página web?

El usuario debe permitir a un navegador conocer la existencia del feed RSS y su localización, cuando entren y se muestre la página web. Por ejemplo, Firefox mostrará un icono de Feed dentro del campo de URL mientras que Internet Explorer lo hará en la barra de comandos.

2.3. Podcasts

Los podcasts son una forma de feeds RSS que son generalmente archivos de audio o vídeo. Los podcasts son compartidos a través de Internet para que cualquiera pueda descargarlos o suscribirse a ellos. RSS es utilizado como una manera rápida de proporcionar información a los subscriptores acerca de los podcasts y hacerles saber cuando hay disponibles nuevos episodios. Los subscriptores de podcasts (*Podcatchers*) utilizan Internet para conectarse periódicamente al feed RSS y comprobar si existen nuevos episodios. Si un podcatcher ve que hay nuevos episodios disponibles, entonces se los descargará.

Los podcasts no son estrictamente ficheros de audio. Los podcasts pueden contener cualquiera de los cientos de tipos de ficheros multimedia existentes, desde mp3, pdf, avi, etc. La cosa adquiere más interés cuando se combinan varios tipos de contenido. Por ejemplo, incluir con un vídeo la copia del audio y un pdf con las notas del mismo. La publicación de podcasts tiene muchas más implicaciones que simplemente la grabación de programas de radio. Los podcasts son compartidos a través de Internet y cualquiera puede descargarse los episodios o suscribirse al podcast. El protocolo RSS es usado como una manera de dar a los subscriptores información acerca de los podcasts, y decirles cuándo hay nuevos episodios disponibles. Los subscriptores utilizan Internet para conectarse periódicamente al feed RSS y comprobar la existencia de nuevos episodios.

Capítulo 3

DroidCatcher: Gestor de Podcast

En este capítulo se hace una descripción de DroidCatcher, una aplicación cuyo propósito es gestionar y administrar suscripciones a podcasts, descarga de capítulos, búsqueda de suscripciones, etc. Además, se detallan las clases que componen la aplicación y se da una breve descripción de ellas. En el próximo capítulo, Desarrollo e implementación de DroidCatcher, se entrará en profundidad en los detalles del desarrollo e implementación de la aplicación

Mediante este desarrollo se busca ilustrar de una forma más práctica las características principales que ofrece Android y pretende además servir como ejemplo para la creación de otras aplicaciones. Algunos de los aspectos más interesantes de Android y que se explican con esta aplicación son los siguientes:

- Uso de componentes Activity.
- Uso de componentes Service.
- Solicitudes a través de Intents.
- Comunicaciones por HTTP.
- Uso de la base de datos SQLite.
- Composición del archivo "AndroidManifest.xml".
- Uso de Notificaciones al usuario.
- Uso de interfaces remotas con AIDL.

- Composición de interfaces de usuario, tanto con código como con XML.
- Declaración y uso de recursos externos.
- Composición gráfica de elementos en pantalla.
- Gestión de opciones de menú.
- Conexión a Internet.

3.1. Descripción de DroidCatcher

DroidCatcher es una aplicación que permite al usuario hacer una completa administración y gestión de suscripciones a servicios de podcasts mediante RSS. Mediante esta aplicación se permite al usuario realizar las siguientes tareas:

- Realizar nuevas suscripciones.
- Buscar nuevos podcasts.
- Descarga de capítulos.
- Reproducción de capítulos.

Para la realización de la aplicación se ha hecho uso de los componentes más comunes de Android, además de algunas características especiales como los Servicios de Notificación, mediante las cuales se comunica al usuario cuando hay nuevos capítulos disponibles.

DroidCatcher permite la descarga de capítulos a través de Internet mediante el uso de la conexión WiFi o bien mediante el uso de la tecnología 3G. Como es lógico, no es condición necesaria establecer una conexión a Internet de forma permanente para hacer uso de DroidCatcher. Si no se tiene acceso a una red, se podrá hacer uso de la aplicación para el resto de funcionalidades como la reproducción o la gestión de los capítulos ya disponibles en el dispositivo. Sin embargo, la conexión a la red es imprescindible para buscar nuevos podcasts, suscribirse a ellos y descargar capítulos. También será necesaria para que el servicio de notificaciones funcione correctamente. A lo largo del capítulo se explicará como se configura la aplicación para tener acceso a Internet.

NOTA: la aplicación DroidCatcher ha sido desarrollada y probada utilizando la versión 2.2 (API nivel 8) de Android. La aplicación no puede ser instalada en versiones anteriores del sistema.

Figura 3.1: Pantalla principal de la aplicación.

3.2. Funcionamiento de la aplicación

Una vez iniciada la aplicación DroidCatcher desde el escritorio del dispositivo Android, se mostrará la pantalla principal de la aplicación.

Dicha pantalla se encarga de mostrar al usuario los paneles con las opciones principales de la aplicación: podcasts, categorías, nuevas suscripciones, descargas y configuración. Como se muestra en la figura 3.1, el panel por defecto es el de los podcasts a los cuales el usuario está subscrito .

Al iniciarse la aplicación, se comprueba si se está reproduciendo algún episodio y, de ser así, se mostrará un acceso directo para que el usuario pueda acceder al reproductor rápidamente si así lo desea (Figura 3.2).

3.2.1. Descripción de la funcionalidad de la aplicación

En este apartado se hará una pequeña descripción de los diferentes casos de uso que se pueden dar en la aplicación. Mediante esta breve introducción se pretende ofrecer al usuario una visión global de lo que realmente puede hacer la aplicación.

- Listado de Podcast: es la pantalla de inicio de la aplicación. En ella, se muestra un listado con todos los podcasts a los que está suscrito el

Figura 3.2: Pantalla principal de la aplicación cuando se está reproduciendo un capítulo.

usuario. Para cada Podcast se muestra su nombre, una imagen descriptiva y la fecha de la última actualización.

- Listado de Categorías: se muestra al usuario un listado con todas las categorías en las que pueden estar clasificados los podcasts. De esta manera se proporciona al usuario un rápido acceso a aquellos podcasts de una temática en concreto, mejorando la experiencia final. Fig 3.3.

- Nueva Suscripción: se podrá realizar a su vez de dos maneras.

  - Introduciendo directamente la URL de la suscripción: al usuario se le presenta un campo de texto donde se deberá introducir la dirección URL de la suscripción. El usuario deberá conocer de antemano dicha suscripción. Figura 3.4.
  - Utilizando el buscador de podcast: el usuario podrá buscar podcasts en Internet usando el buscador predeterminado de DroidCatcher. El buscador de podcast es una herramienta muy útil para la aplicación pues permite, mediante una interfaz clara y sencilla, buscar podcast usando palabras clave. Para ello no hay más que introducir el término por el cuál se quiera realizar la

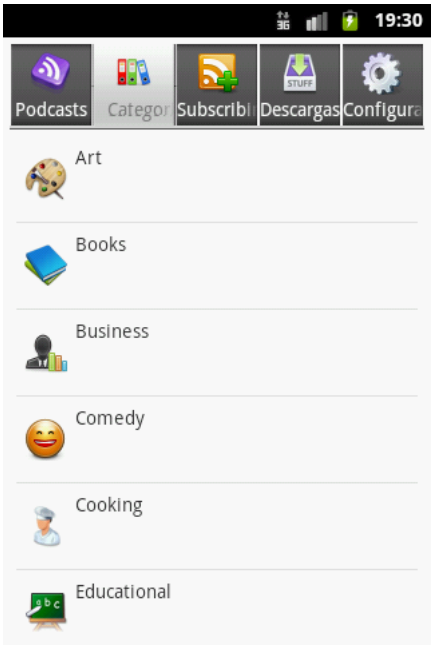

Figura 3.3: Listado con las categorías de Podcast.

Figura 3.4: Suscripción mediante URL.

Figura 3.5: Búsqueda de podcasts de cine

búsqueda y a continuación se mostrará un listado con los resultados coincidentes. En dicho listado se muestra Título, Descripción y un botón para realizar la suscripción automáticamente. Dicho buscador realiza peticiones HTTP a la página de búsqueda `www.digitalpodcast.com` o bien a la página `www.podsonoro.es` (a elección del usuario) y presentará los resultados al usuario de forma resumida. El usuario puede introducir una palabra clave para realizar su búsqueda. La información presentada contendrá una pequeña descripción de la suscripción y una opción de "suscribirse". Figura 3.5.

- Descargas Actuales: se muestra al usuario un listado con las descargas que se están realizando en este momento. Dicho listado contiene información tal que el título del episodio, el nombre del podcast y una barra de progreso con el estado de la descarga actual. Dichas descargas podrán ser canceladas desde este mismo listado. Figura 3.6.

- Configuración de la Aplicación: desde esta página se podrán configurar diversos parámetros de la aplicación (Figura 3.7). Dichos parámetros son:

- Listado de Episodios de un podcast: se accede seleccionando un pod-

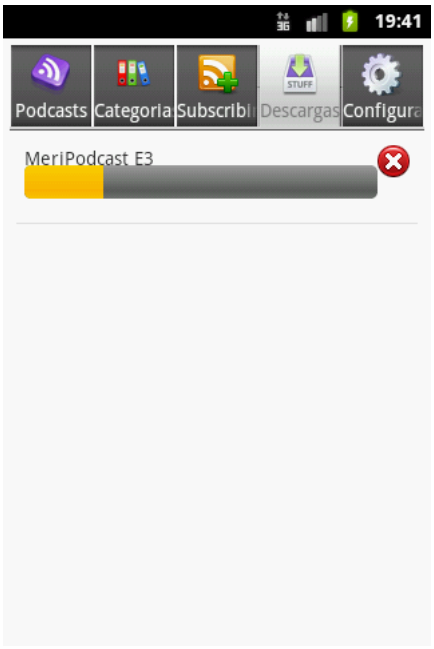

Figura 3.6: Pantalla con las descargas actuales

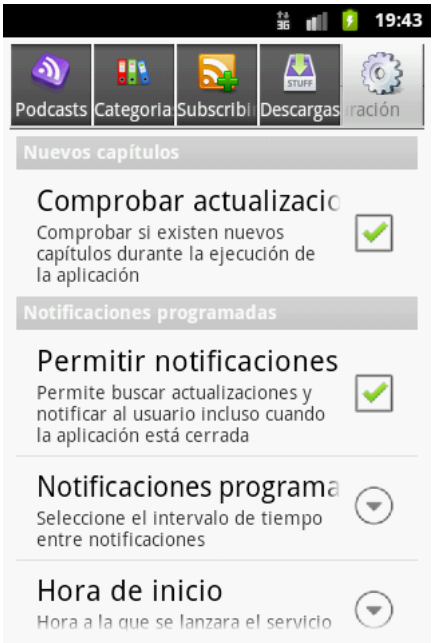

Figura 3.7: Preferencias de DroidCatcher

Figura 3.8: Listado de capítulos de un podcast

cast. Se muestra al usuario una lista con todos los capítulos que han sido descargados de el podcast seleccionado. Cada episodio muestra información de su contenido, fecha de publicación, imagen descriptiva. Además, se muestra información al usuario sobre si el episodio ha sido ya escuchado o si ha sido marcado como favorito por el usuario. Desde esta página se podrá seleccionar un episodio para ser escuchado, o bien para ser borrado. Además, desde esta vista se podrá acceder a la descarga manual de episodios. Figura 3.8.

- Descarga Manual de Episodios: listado con los episodios actualmente disponibles para ser descargados. Muestra información facilitada por la página de suscripción, tal como la fecha de publicación, descripción, nombre, etc.

- Importado de Episodios desde tarjeta de memoria: también es posible importar episodios que se tengan almacenados en tarjetas de memoria. Para ello, bastaría con realizar una pulsación prolongada sobre el episodio en concreto que se quiera importar y seleccionar la opción de "importar". Acto seguido, solo habrá que navegar por el sistema de ficheros de Android para seleccionar el archivo deseado. Figuras 3.9 y 3.10.

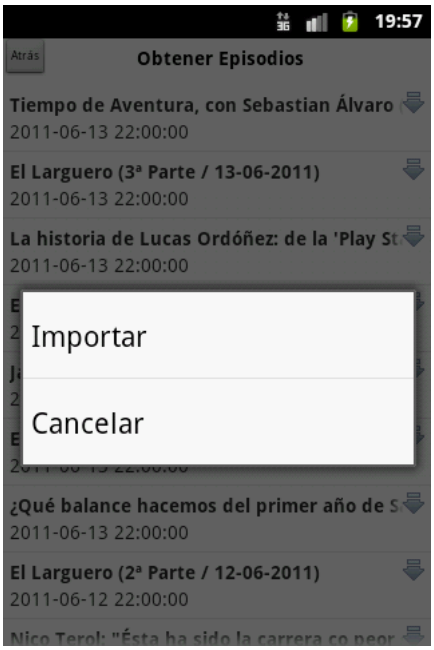

Figura 3.9: Importar un episodio

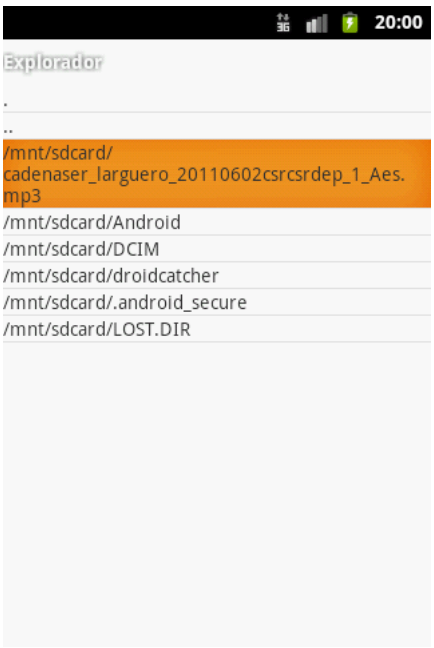

Figura 3.10: Explorador de archivos para importar un episodio

Figura 3.11: Reproductor de capítulos

- Reproducción de Episodios: DroidCatcher proporciona un reproductor con una sencilla interfaz para que el usuario escuche los episodios descargados. Además de las opciones clásicas de reproducción (Play, Pause, Stop, Next y Prev), se proporciona información detallada del capítulo, así como la posibilidad de marcar el episodio como favorito. Figura 3.11

- Servicio de búsqueda y notificación de capítulos disponibles, UpdateService: es un servicio configurable por el usuario desde la pestaña "configuración". Si el servicio está activo será el encargado de buscar episodios disponibles para los podcasts a los cuales el usuario esté suscrito. El usuario podrá activar/desactivar el servicio y, además, podrá configurar el intervalo de tiempo que transcurrirá entre dos actualizaciones. Figuras 3.12 y 3.13. Además, se podrá configurar el servicio para que descargue automáticamente los nuevos episodios encontrados. Cuando el servicio detecte que hay episodios nuevos disponibles, mandará una notificación al usuario usando para ello la clase `NotificationManager`. Figuras 3.14 y 3.15.

Cabe destacar que en cada una de las pantallas que ofrece la aplicación, se mostrará un título dinámico, con información relativa a la ventana donde nos

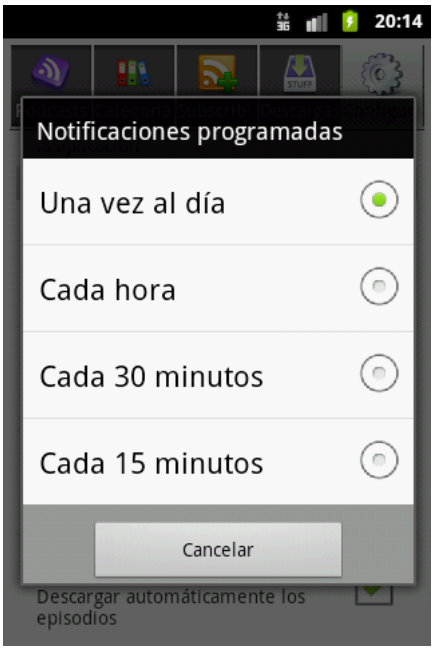

Figura 3.12: Configuración servicio de actualización

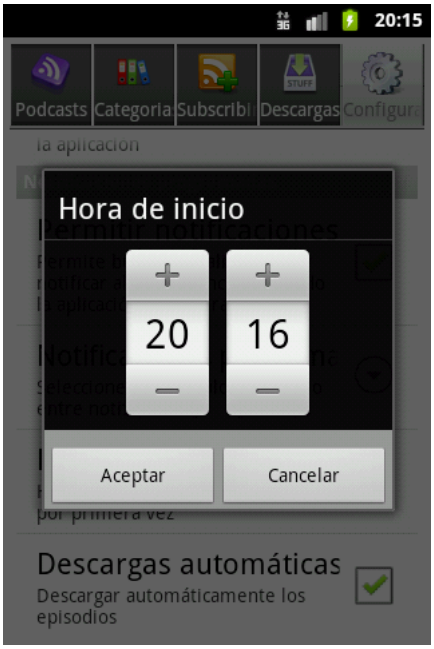

Figura 3.13: Hora de inicia del servicio

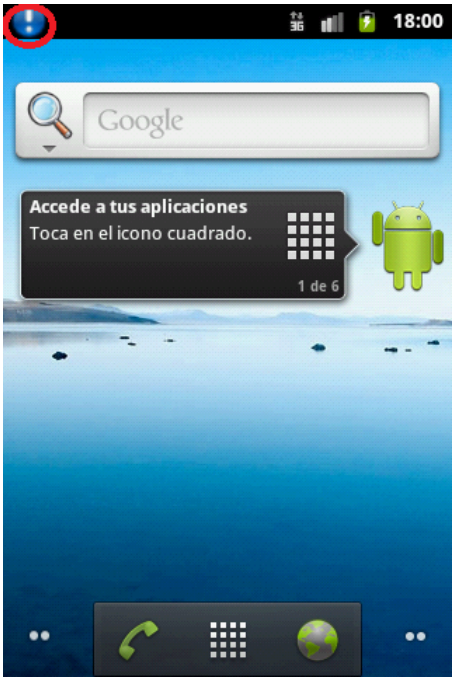

Figura 3.14: Notificación de actualizaciones disponibles.

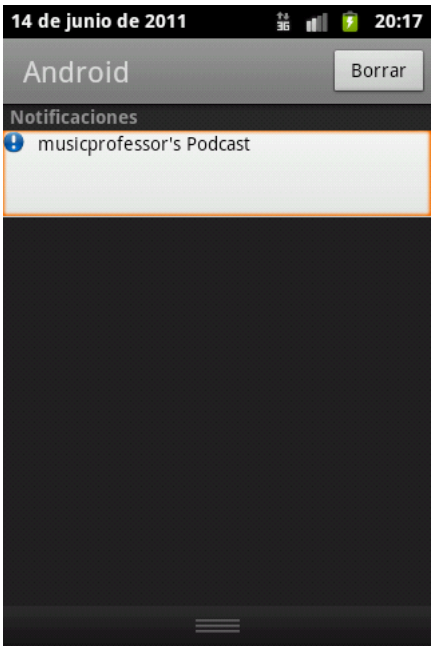

Figura 3.15: Notificación de actualizaciones disponibles.

encontramos. Además, dado que la aplicación está pensada para dispositivos táctiles, habrá a disposición del usuario un botón para volver atrás. Si se está reproduciendo un episodio, también habrá un botón con acceso directo al reproductor de la aplicación.

3.3. Descripción de clases

3.3.1. Diagrama de clases de DroidCatcher

Las clases utilizadas para resolver un determinado problema, sus atributos y métodos, la visibilidad que de estos tienen las demás clases, así como las relaciones que existen entre ellas y sus colaboraciones, constituyen el modelo de clases. Mediante este tipo de modelos se expresa, con mayor o menor nivel de detalle, la futura implementación del sistema, así como permite dar una idea bastante cercana a la forma en la que se ha abordado el problema.

En el presente apartado se ofrece al lector una breve descripción de cuál es el modelo de clases utilizado en la aplicación DroidCatcher. El objetivo principal es que el lector se haga una idea global de la implementación de DroidCatcher y que será explicada en profundidad en capítulos posteriores. Por todo ello, el modelo de clases ofrecido no incluye más que clases, atributos, métodos, y la visibilidad de estos, con vistas a no perder al lector con aspectos demasiado complejos.

En las figuras 3.16, 3.17 y 3.18 se enseña el diagrama correspondiente al modelo de clases utilizado, siguiendo el estándar UML 2.0 [35].

A continuación se ofrece una breve explicación de cada una de las clases expuestas en el diagrama anterior. Para facilitar la rápida localización de información, se han creado una serie de paquetes con el fin de tener un código más ordenado y de fácil comprensión.

- **com.google.android.droidcatcher**: dentro de este paquete se encuentran las clases que representan a los miembros de la aplicación, podcasts, categorías y episodios:
  - **Podcast**: representa la información correspondiente a un podcast.
  - **Episode**: clase que se corresponde con un episodio, con información relativa a él y una referencia al podcast al que pertenece.
  - **Category**: representa una categoría de podcast dentro de la aplicación.
- **com.google.android.droidcatcher.activity**: se podría decir de este paquete que es el principal dentro de la aplicación. En él se encuentran

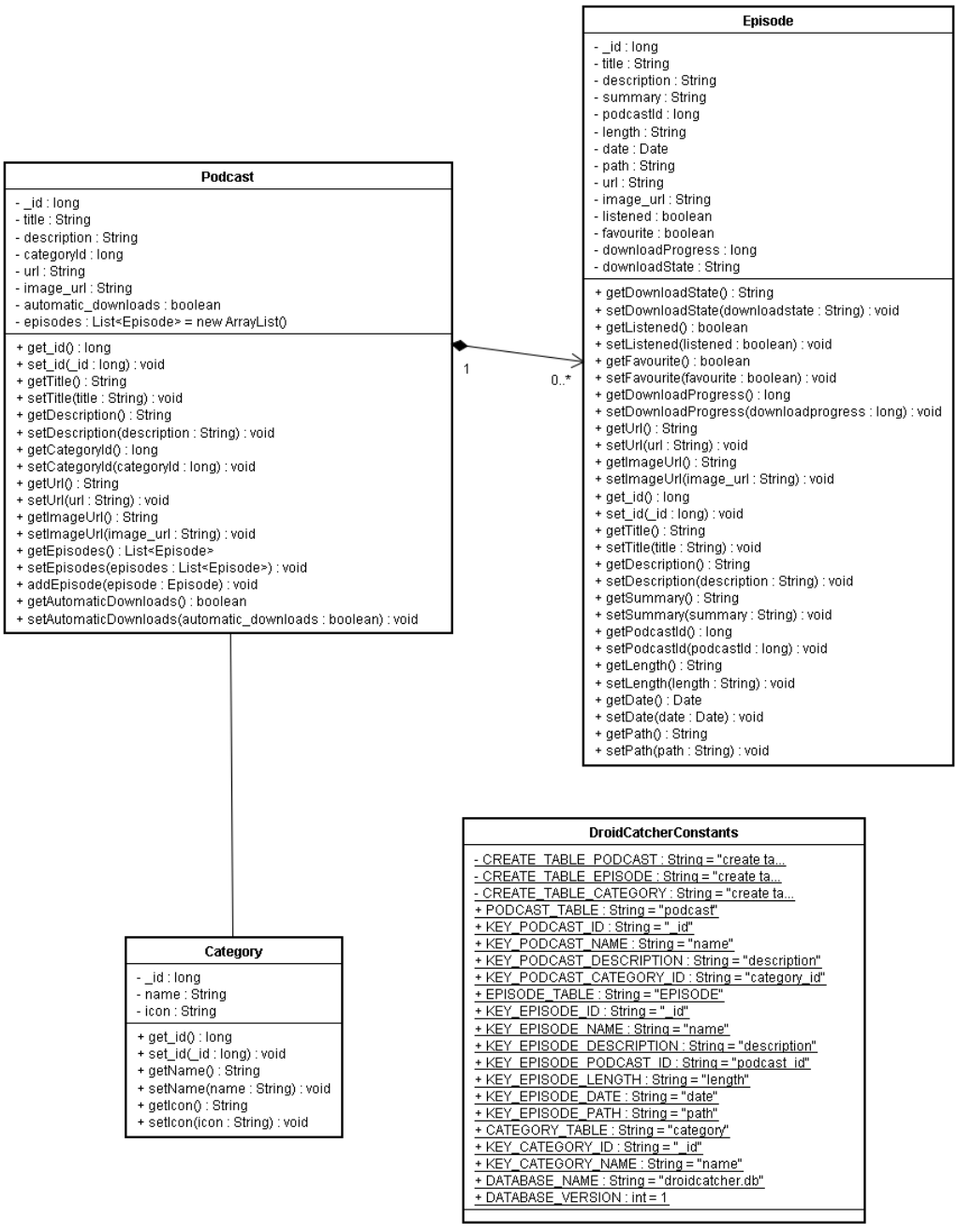

Figura 3.16: Diagrama UML de las clases utilizadas.

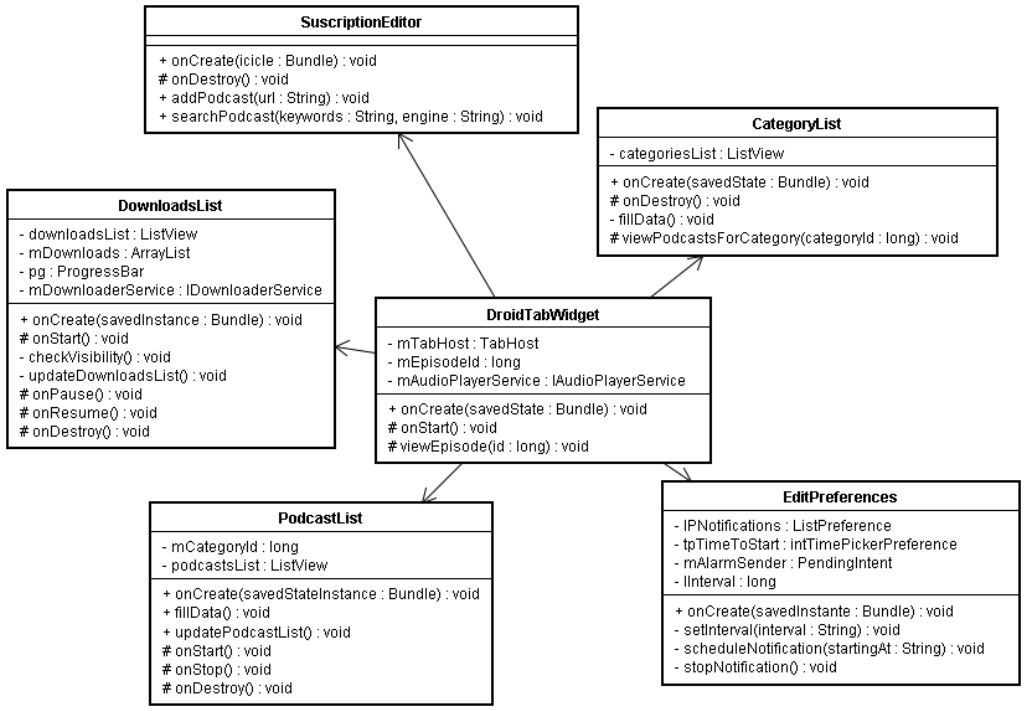

Figura 3.17: Diagrama UML de las clases utilizadas.

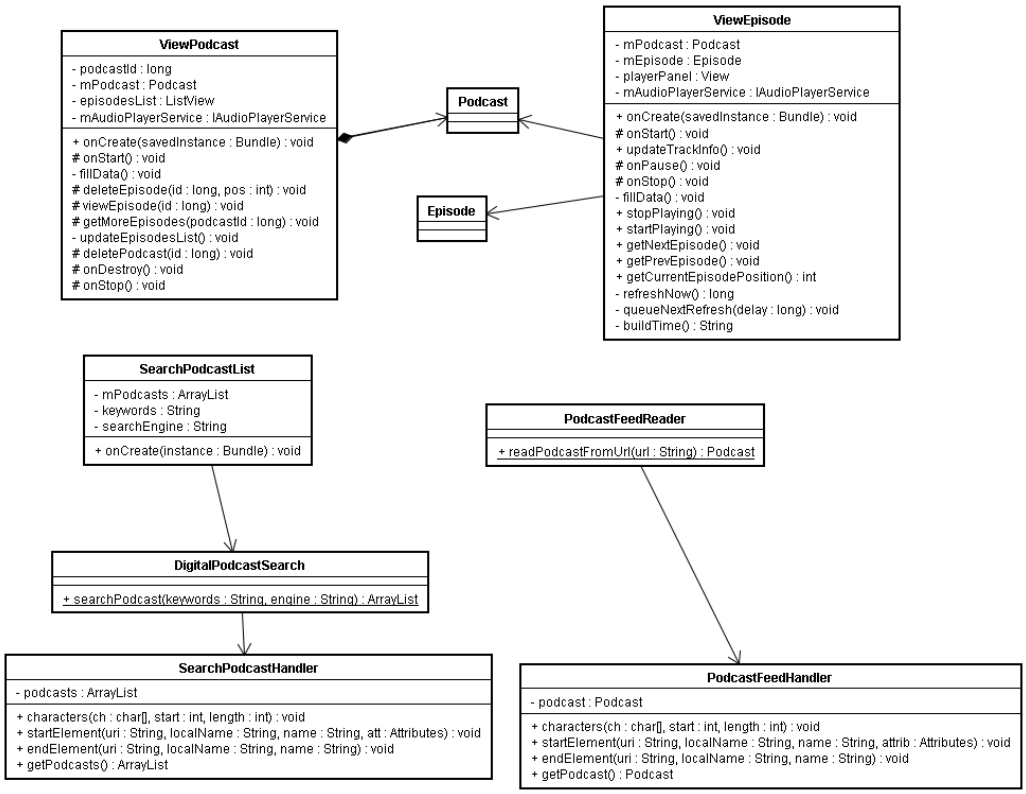

Figura 3.18: Diagrama UML de las clases utilizadas.

todas la actividades que componen la aplicación, así como los `Adapters` que se encargan de rellenar con la información obtenida de BBDD los distintos elementos de la pantalla. Todas las actividades extienden a la clase `Activity`.

- **DroidTabWidget**: se trata del punto de partida de la aplicación. Es la clase principal y la que coordina la mayor parte de la ejecución de la aplicación. Mediante `DroidTabWidget`, que extiende la clase `TabActivity`, se muestran al usuario los paneles con las opciones principales de la aplicación: podcasts, categorias, nuevas suscripciones, descargas y configuración.
- **PodcastList**: esta clase extiende a `ListActivity`. Se encarga de mostrar en pantalla un listado con los distintos podcasts a los que está suscrito el usuario. Cada uno de ellos será seleccionable para obtener más información acerca de él.
- **CategoryList**: muestra en pantalla un listado con las diferentes categorías en las que se pueden ordenar los podcasts dentro de la aplicación. La clase utiliza un adaptador personalizado (`CursorCategoryAdapter`) para mostrar estas categorías en la pantalla. Estas categorías son predefinidas y abarcan un amplio rango de posibilidades. Cada categoría puede ser seleccionada para mostrar los podcasts pertenecientes a la misma.
- **SubscriptionEditor**: esta clase muestra al usuario un formulario para realizar una nueva suscripción a un podcast. Es una clase que extiende de `Activity`.
- **DownloadsList**: muestra un listado con los capítulos que se encuentran actualmente descargándose o en estado de espera. Muestra una barra de progreso que indica qué porcentaje de episodio se ha descargado, así como un botón para cancelar dicha descarga.
- **EditPreferences**: permite al usuario configurar determinados aspectos de la aplicación. Desde el tipo de almacenamiento usado por la aplicación hasta la frecuencia con la que se comprueba la existencia de nuevos capítulos en el servidor.
- **ViewPodcast**: muestra al usuario la información detallada de un podcast al que se está suscrito además de un listado de los episodios que han sido descargados. Desde esta actividad se pueden borrar capítulos de manera individual o el podcast completo.
- **GetMoreEpisodesList**: actividad que crea un listado de episodios disponibles en el servidor y que todavía no han sido descar-

gados por el usuario. Desde esta vista se puede iniciar la descarga de capítulos.

- **ViewEpisode**: actividad muy importante dentro de la aplicación pues es desde la cuál se inicia la reproducción de un capítulo descargado. Además, muestra información detallada del mismo, pudiéndose establecer el capítulo como favorito.

- **com.google.android.droidcatcher.animation**: paquete donde se encuentran las clases especiales para la implementación de animaciones personalizadas.

  - **Rotate3dAnimation**: crea una animación sobre un elemento de la interfaz de usuario. Rota la vista en el eje Y de coordenadas entre dos ángulos específicos. Además, añade una traslación sobre el eje Z para potenciar el efecto.

- **com.google.android.droidcatcher.db**: paquete donde se encuentran las clases que gestionan la base de datos.

  - **DroidCatcherDB**: clase que gestiona el acceso a la base de datos SQLite para almacenar o consultar los datos tanto de los podcast, como de Episodios y Categorías de DroidCatcher.

- **com.google.android.droidcatcher.exception**: excepciones propias de la aplicación.

  - **DownloaderException**: excepción provocada por un fallo en la descarga de episodios.

- **com.google.android.droidcatcher.preference**: paquete donde se encuentran agrupadas clases específicas para la creación de las preferencias de la aplicación.

  - **TimePickerPreference**: preferencia personalizada que muestra un cuadro de diálogo para escoger una hora del día.

- **com.google.droidcatcher.service**: aquí se encuentran las clases e interfaces AIDL necesarios para la definición de los servicios de DroidCatcher.

  - **AudioPlayerService**: servicio de audio de DroidCatcher. Se encarga de la reproducción de los episodios y de que ésta no finalice incluso cuando se salga de la aplicación.

  - **DownloaderService**: servicio de descargas. Gestiona la descarga de nuevos capítulos a través de Internet.
  - **UpdateService**: servicio de actualizaciones de podcasts. Cuando dicha funcionalidad está activa, el servicio se encarga de buscar actualizaciones (nuevos episodios) para los podcasts a los que está suscrito el usuario, notificándole en caso afirmativo. Opcionalmente, puede iniciar la descarga de los nuevos capítulos encontrados.
  - **ServiceUtils**: clase de utilidades para los servicios, gestiona tareas como la creación y el borrado de los mismos.

- **com.google.android.droidcatcher.reader**: dentro de este paquete se encuentran las clases encargadas del procesamiento de datos XML obtenidos de Internet, de los cuales la aplicación extrae la información relativa a las suscripciones del usuario.

  - **DigitalPodcastSearch**: esta clase es capaz de realizar una búsqueda de podcasts en función de las palabras clave que el usuario introduzca. Actualmente, la aplicación funciona con dos motores de búsqueda ya implementados, *DigitalPodcast* y *PodSonoro*. Con la ayuda de `SearchPodcastHandler`, crea un listado de podcasts que pueden coincidir con la búsqueda del usuario.
  - **SearchPodcastHandler**: clase que parsea la información devuelta por el motor de búsqueda (en formato XML)
  - **PodcastFeedReader**: esta clase realiza suscripciones a podcasts, a partir de una dirección de Internet proporcionada por el usuario. Realiza una conexión HTTP y, con la ayuda de `PodcastFeedHandler`, almacena los datos relativos a la suscripción en la base de datos.
  - **PodcastFeedHandler**: clase que parsea la información devuelta como consecuencia de una petición HTTP a una página proporcionada por el usuario.

- **com.google.android.droidcatcher.util**: paquete de utilidades.

  - **DateUtil**:clase que contiene métodos de utilidades para los formatos de fechas.
  - **DownloaderUtil**: clase para la descarga de archivos.
  - **StorageUtils**: clase que proporciona información útil acerca del sistema de almacenamiento de Android.

    - **SystemUtils**: clase con métodos que proporcionan información acerca del sistema, como conexiones de red.

- **com.google.android.droidcatcher.view**: clases que representan vistas personalizadas para la aplicación.

    - **PlayerPanel**: vista personalizada para representar el panel de control del reproductor de episodios.

Capítulo 4

Desarrollo e implementación de DroidCatcher

4.1. Introducción

Una vez expuestos los objetivos y características principales de la aplicación, así como las decisiones tomadas en cuanto a su diseño, a continuación se explican en este apartado los aspectos relacionados con su implementación.

El objetivo de este capítulo no es sólo mostrar el código más relevante de DroidCatcher y explicar el funcionamiento de éste, sino servir también como manual genérico sobre el desarrollo de aplicaciones sobre la plataforma Android. De este modo, se contarán desde los detalles más básicos para el desarrollo de la aplicación hasta las particularidades y casos especiales encontrados en DroidCatcher. Así mismo, los detalles de implementación mencionados podrán servir de ayuda a otros desarrolladores que se propongan ampliar y mejorar en un futuro las capacidades de DroidCatcher, o simplemente sirva de ayuda para la creación de nuevas aplicaciones totalmente diferentes en funcionalidad, pero similares en el desarrollo genérico.

Para el desarrollo del proyecto, el autor ha hecho uso frecuente de la amplia comunidad de desarrollares que existen actualmente de la plataforma. Sitios web como **Andev Community** [52], **HelloAndroid Tutorials** [53] y **Statck Overflow** [54] son sólo unos ejemplos de comunidades de desarrollo Android donde se ofrecen tutoriales de todo tipo, así como soluciones a problemas frecuentes encontrados durante el desarrollo de aplicaciones Android.

4.2. AndroidManifest

El fichero de manifiesto de DroidCatcher no es excesivamente complicado, pero, como ya se dijo anteriormente, es obligatorio en toda aplicación Android y ésta no iba a ser una excepción. A continuación se verán algunos puntos importantes de este fichero, como la declaración del paquete JAVA al que pertenece la aplicación, la declaración de algunos componentes como `Activity, Service o BroadcastReceiver` y también la declaración de los permisos de la aplicación.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
      package="com.google.android.droidcatcher.activity"
      android:versionCode="1"
      android:versionName="1.0.0">
```

En este fragmento de código se declara el paquete al que pertenece la aplicación, así como el número de versión del código y el nombre de la versión.

A continuación se muestra la declaración del Activity principal de la aplicación, esto es el que se lanzará cuando la aplicación arranque:

```
<activity android:name=".DroidTabWidget"
          android:label="@string/app_name"
          android:theme="@android:style/Theme.Light.
              NoTitleBar"
          android:debuggable="true">
    <intent-filter>
          <action android:name="android.intent.action.MAIN"
              />
          <category android:name="android.intent.category.
              LAUNCHER" />
    </intent-filter>
</activity>
```

Declaración de un servicio:

```
<service android:name="com.google.android.droidcatcher.
    service.DownloaderService"></service>
```

Declaración de un `BroadcastReceiver`

```
<receiver android:enabled="true" android:name=".
    BootUpReceiver"
     android:permission="android.permission.
        RECEIVE_BOOT_COMPLETED">
```

```
3       <intent-filter>
4           <action android:name="android.intent.action.
                BOOT_COMPLETED" />
5           <category android:name="android.intent.category.
                DEFAULT" />
6       </intent-filter>
7   </receiver>
```

Esta parte es interesante, pues se está declarando un `BroadcastReceiver` que responderá ante un determinado `Intent`, en este caso `android.intent.action.BOOT_COMPLETED`. Con esto lo que se consigue es que cuando el sistema operativo haya arrancado completamente, se ejecutará el método `onReceive()` del `BroadcastReceiver` declarado, (`BootUpReceiver`).

Permisos de la aplicación:

```
1   <uses-permission android:name="android.permission.INTERNET"
        />
2   <uses-permission android:name="android.permission.
        RECEIVE_BOOT_COMPLETED" />
3   <uses-permission android:name="android.permission.
        ACCESS_NETWORK_STATE" />
```

La aplicación necesita permisos para acceder a Internet, para acceder al estado de la red y también para recibir una notificación cuando el sistema haya terminado de arrancar.

4.3. Punto de partida de la aplicación: Interfaz de Usuario

Este bloque está estrechamente relacionado con las actividades de la aplicación

Como se ha visto anteriormente, existen dos maneras de declarar la interfaz de usuario en una aplicación Android: se puede definir la interfaz de usuario en un fichero XML, definiendo las vistas y grupos de vistas utilizando vocabulario específico que Android proporciona para tal efecto, o bien se pueden instanciar estas vistas en tiempo de ejecución programáticamente. Para el diseño de DroidCatcher se ha optado por la primera opción, ya que de esta forma no sólo resulta muy sencillo y claro definir la interfaz de usuario sino que además permite tener separados los aspectos visuales de la aplicación de los funcionales.

Dentro de la carpeta *res/layout/* se encuentran los archivos .xml que forman parte de la interfaz de usuario. Algunos de ellos se corresponden con un único Activity (caso de view_episode.xml), mientras que otros pueden agrupar varios (main.xml).

Básicamente, todos los elementos XML que forman parte de la interfaz de usuario de DroidCatcher tienen una estructura muy parecida. Cada uno de ellos, llamados *layouts*, declaran una parte de la interfaz que normalmente representa no sólo el aspecto visual de una Activity, sino también la forma de interactuar con ella. Cada *layout* define de qué forma se dispondrán los elementos en pantalla, además de su posición, tamaño, etc. Para entender mejor el funcionamiento de estos ficheros de la interfaz de usuario, se explicarán con detalle los más relevantes de la aplicación. Una vez entendidos éstos, el resto son simplemente extensiones o simplificaciones de los primeros, por lo que se podrían entender con un simple vistazo a los mismos.

4.3.1. Navegación por pestañas

En primer lugar se va a ver el principal *layout* de la aplicación, que es el que se muestra al abrir la aplicación.

main.xml

```
1
2  <?xml version="1.0" encoding="utf-8"?>
3  <TabHost xmlns:android="http://schemas.android.com/apk/res/
       android"
4      android:id="@android:id/tabhost"
5      android:layout_width="fill_parent"
6      android:layout_height="fill_parent">
7      <LinearLayout
8          android:orientation="vertical"
9          android:layout_width="fill_parent"
10         android:layout_height="fill_parent"
11         android:padding="5dp">
12         <TabWidget
13             android:id="@android:id/tabs"
14             android:layout_width="fill_parent"
15             android:layout_height="wrap_content"
16             />
17         <FrameLayout
18             android:id="@android:id/tabcontent"
19             android:layout_width="fill_parent"
20             android:layout_height="fill_parent"
21             android:padding="5dp"/>
22     </LinearLayout>
23     <include layout = "@layout/now_playing_layout" />
```

```
24 </TabHost>
```

Un `TabHost` es un contenedor para una vista con pestañas [55]. Este objeto tiene dos hijos: una serie de etiquetas para que el usuario navegue por ellas; y un objeto `FrameLayout` con el contenido de cada pestaña.

Este es el layout que muestra en pantalla pestañas de navegación entre las distintas Activities creadas previamente. Dentro de `TabHost`, es obligatorio que existan otros dos elementos, `Tabwidget` y `FrameLayout`. `TabWidgets` para las pestañas y `FrameLayout` para el contenido de cada una de ellas, generalmente un `Activity`. Para posicionar estos dos elementos se utiliza otro elemento, `LinearLayout`. El elemento `FrameLayout` es el utilizado para mostrar el contenido de cada pestaña. En un principio va vacío, ya que el elemento `TabHost` meterá cada `Activiy` automáticamente dentro.

Nótese que los identificadores de `TabWidget` y `FrameLayout` son "tabs" y "tabcontent" respectivamente. Estos nombres son obligatorios, de modo que el objeto `TabHost` pueda obtener las referencias a ellos para rellenarlos correctamente.

En las figuras 4.2 y 4.1 se muestra una representación gráfica de la jerarquía de vistas resultado de la utilización de un `TabHost` para la interfaz de usuario. Como se puede observar, aparecen las cinco pestañas pertenecientes a la vista, pero solamente aparece el contenido de una de ellas (así como los hijos de ésta), ya que en el momento de la captura sólo se muestra el contenido de la pestaña en la que la aplicación se encuentra, en este caso, el listado de podcasts. Para crear una interfaz de usuario en Android no basta con definir un layout. Con este primer paso lo que se ha hecho es definir la estructura de cómo se va a mostrar o cómo va a interactuar una Activity con el usuario, pero para que esto tenga sentido hacen falta dos cosas más:

- Hay que establecer la relación entre un `Activity` y su interfaz de usuario.
- Hay que "rellenar" con datos esta interfaz.

4.3.2. Relación Activity layout

Para que se muestre en pantalla, este *layout* tendrá que relacionarse con una Activity. Este paso es muy sencillo y simplemente bastará con indicar qué fichero contiene la interfaz gráfica para una Activity en el momento en que se esté creando dicha actividad:

```
1 public void onCreate(Bundle savedInstanceState) {
2         super.onCreate(savedInstanceState);
```

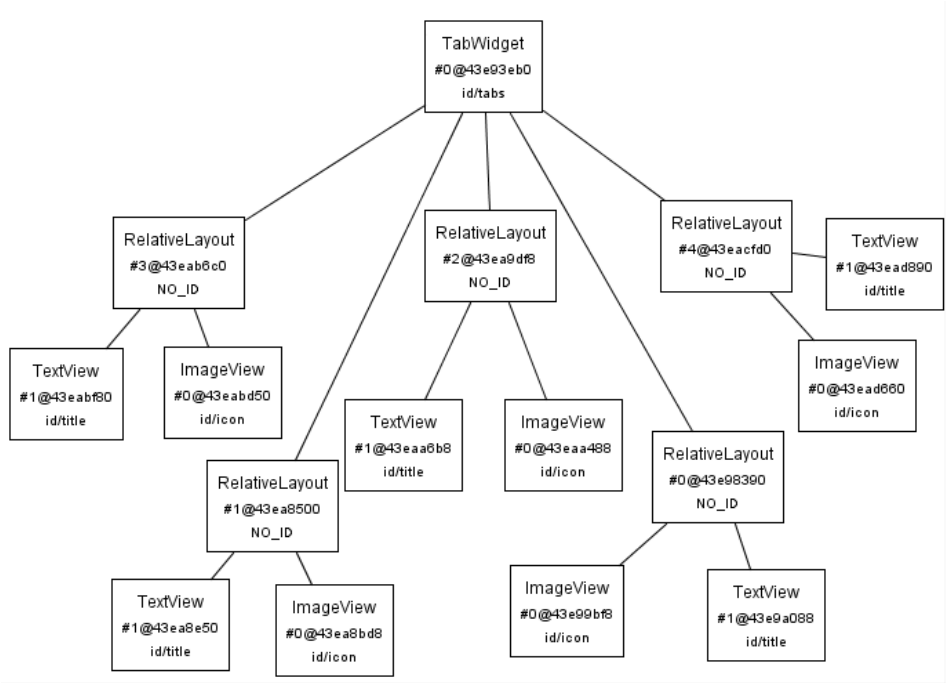

Figura 4.1: Jerarquía de vistas. Tabs.

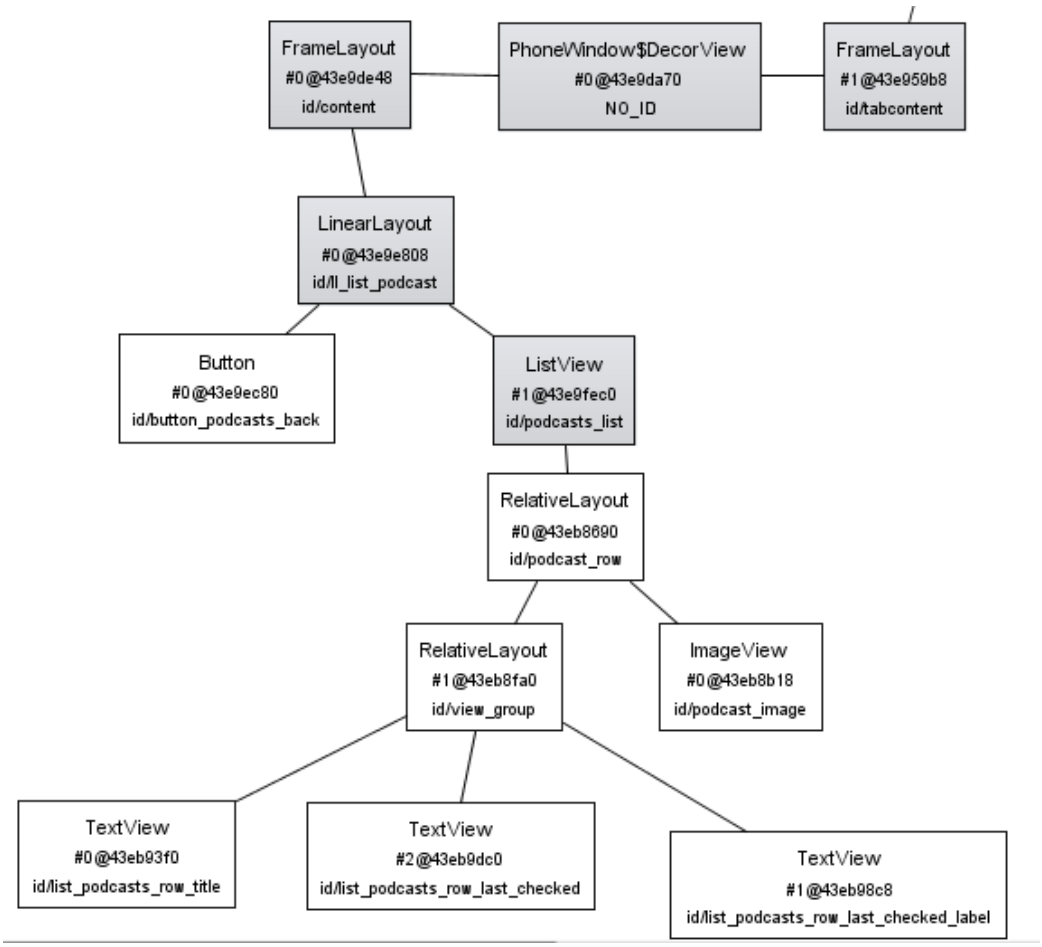

Figura 4.2: Jerarquía de vistas. Tabcontent.

```
3           setContentView(R.layout.main);
4       ...
5       ...
6       ...
```

El siguiente paso a realizar es completar con datos la Activity que se está mostrando al usuario. Normalmente, lo que se querrá hacer es obtener datos del algún sitio (generalmente una base de datos) y mostrarlos al usuario de una determinada forma. Ya se ha visto que el aspecto visual de cómo se muestran esos datos lo determina el layout, ahora queda por ver cómo se rellena la pantalla que ve el usuario con los datos que a él le son interesantes.

El caso de la pantalla principal de la aplicación es un tanto especial, ya que se trata no sólo de mostrar información al usuario acerca de podcasts, sino también de mostrarle el resto de operaciones que se pueden llevar a cabo en la aplicación. Primero se detallará cómo se crea este menú de navegación de la aplicación para luego pasar a explicar cómo se rellena cada Activity con información concreta acerca de los diversos contenidos de la aplicación.

DroidTabWidget extiende a la clase `TabActivity`. Esta es la actividad que se llama primeramente cuando se inicia la aplicación. Cada panel que se presenta al usuario es una actividad en sí misma. Un `TabWidget` ofrece la posibilidad de presentar al usuario de una forma fácil una interfaz con pestañas con las cuales se podrá navegar entre distintas vistas. El método `getTabHost()` devuelve una referencia al objeto `TabHost` que se ha creado en la definición de la interfaz de usuario (main.xml). Con dicho objeto `TabHost`, se llama al método `addTab()` por cada una de las pestañas que se quieran añadir. Cada vez que se cree una pestaña, se pasa como parámetro un objeto de tipo `TabSpec` que se usará para indicar el texto a mostrar en la pestaña, el icono y el `Intent` que se usará para lanzar la actividad asociada a la pestaña.

```
1
2   TabHost.TabSpec spec;  // Reusable TabSpec for each tab
3       Intent intent;  // Reusable Intent for each tab
4
5       // Create an Intent to launch an Activity for the tab (to
            be reused)
6       intent = new Intent().setClass(this, PodcastList.class);
7       intent.putExtra("menu", false);
8       // Initialize a TabSpec for each tab and add it to the
            TabHost
9       spec = tabHost.newTabSpec("podcasts").setIndicator("
            Podcasts",
10                      res.getDrawable(R.drawable.
                            feed_icon_purple))
11                  .setContent(intent);
```

```
12      tabHost.addTab(spec);
13  }
```

Una vez creadas todas las pestañas se establece una para mostrar por defecto. En este caso será `PodcastList`, la Activity encargada de mostrar el listado de podcasts.

```
1  tabHost.setCurrentTabByTag("podcasts");
```

4.3.3. Aspecto y funcionamiento de cada pestaña

Hasta este punto se ha explicado cómo se crea el aspecto visual global de la aplicación, es decir, las pestañas de navegación y también cómo se asocian las distintas pestañas con Activities. Queda por ver cómo cada una de éstas tiene su propio aspecto visual (definido por su propio layout) y cómo son capaces de mostrar al usuario datos obtenidos de una base de datos. Para continuar con el ejemplo visto arriba, a continuación se explicará todo esto para la actividad encargada de mostar el listado de podcasts al usuario, `PodcastList`.

4.3.4. layout de PodcastList

El modelo gráfico de PodcastList parece en principio muy sencillo, aunque como se verá un poco más adelante tiene ciertas complicaciones. El fichero XML que contiene este modelo es **podcasts.xml**:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/
       res/android"
3      android:orientation="vertical" android:layout_width="
           fill_parent"
4      android:layout_height="fill_parent"
5      android:id="@+id/ll_list_podcast">
6      <Button android:id="@+id/button_podcasts_back"
7          style="@android:style/Widget.Button.Small"
8          android:text="@string/button_back"
9          android:textSize="9px"
10         android:layout_width="wrap_content"
11         android:layout_height="30px"
12         android:onClick="myBackHandler"
13         android:visibility="gone"
14         android:layout_alignParentLeft="true"
15         android:layout_alignParentTop="true"
```

```
16        />
17      <ListView
18          android:id="@+id/podcasts_list"
19          android:layout_width="match_parent"
20          android:layout_height="match_parent"
21          android:layout_below="@+id/button_podcasts_back"/>
22  </LinearLayout>
```

Como se puede observar, la Activity encargada de mostrar los podcasts tiene únicamente dos componentes visuales: un botón de vuelta atrás (que además está escondido en un principio) y un elemento de tipo `ListView`, que es una vista que contiene a su vez varios *items* y que los muestra con un scroll vertical. Este objeto de tipo `ListView` es realmente la clave de esta Activity. Por un lado, hay que definir el aspecto visual de las filas del listado (una vez más, con un layout XML) y además, se tendrá que rellenar este listado **dinámicamente** con datos obtenidos de base de datos. Este último paso se realizará dentro del código JAVA, una parte dentro de la propia Activity y la otra dentro de un tipo de clase especial, los **Adapters**.

4.3.5. PodcastList

A continuación se explican detalladamente los aspectos más importantes de la Activity `PodcastList`:

```
1   public void onCreate(Bundle savedInstanceState) {
2       ...
3           setContentView(R.layout.podcasts);
4           mdb = new DroidCatcherDB(this);
5           podcastsList = (ListView)findViewById(R.id.
                podcasts_list);
6           podcastsList.setOnItemClickListener(...);
7       ...
8       ...
9           fillData();
10      }
```

Se puede observar cómo se le dice a la actividad qué fichero es el que define su aspecto visual, en este caso, *podcasts.xml*. Además, se obtienen una referencia a la BBDD y una referencia al objeto `ListView` declarado **dentro de podcasts.xml**. Por ultimo, se llama al método `fillData()`, el cuál se verá a continuación, y que es donde se rellenan los datos de forma dinámica haciendo uso de la clase `Adapter`.

```
1   private void fillData() {
```

```
2       ...
3       ...
4       if(mCategoryId == -1){
5           podcastsCursor = mdb.fetchAllPodcast();
6       }else{
7           podcastsCursor = mdb.fetchAllPodcastForCategory(
                mCategoryId);
8       }
9       startManagingCursor(podcastsCursor);
10      String[] from = new String[] { DroidCatcherDB.
            KEY_PODCAST_TITLE,
11              DroidCatcherDB.KEY_PODCAST_LAST_CHECKED};
12      int[] to = new int[] { R.id.list_podcasts_row_title,
13          R.id.list_podcasts_row_last_checked };
14      /**Creates the adapter to manage the view and then sets
            it to the view */
15      podcastsAdapter = new CursorPodcastListAdapter(this,
16          R.layout.list_podcasts_row, podcastsCursor, from, to)
                ;
17      podcastsList.setAdapter(podcastsAdapter);
18      }
```

Hay varios puntos interesantes en este método:

- Se hace una consulta a la base de datos para obtener los podcast (todos o los pertenecientes a una categoría). El resultado es devuelto en un objeto de tipo `Cursor`.
- Se llama al método `startManagingCursor`, que permite que sea la propia Activity la que se encargue del ciclo de vida del objeto `Cursor` de forma automática. Es decir, si la actividad se para, llamará automáticamente al método `deactivate()`. Si la actividad se vuelve a reiniciar, llamará al método `requery()`. Cuando finalmente la actividad sea destruida, cerrará todos los cursores que esté gestionando.
- Las tres sentencias siguientes son muy importantes. En la primera, se indica qué columnas de la base de datos se quieren obtener. En la segunda, se indica sobre qué componentes gráficos se han de mapear estos datos. Finalmente, en la tercera sentencia se crea una instancia de la clase `Adapter` (`CursorPodcastListAdapter`). Esta clase será la encargada de mantener los datos que forman el listado de podcasts y producir una vista que represente cada item del conjunto de datos. En la creación de la instancia se ha de indicar, entre otras cosas, el fichero donde están definidos los componentes gráficos para cada item de la lista (list_podcast_row) y el cursor con los datos.

4.3.6. CursorPodcastListAdapter

Esta clase extiende de `SimpleCursorAdapter`: se trata de un sencillo adaptador que mapea columnas de un cursor de datos a elementos gráficos tales como campos de texto (`TextViews`) o imágenes (`ImageViews`). Se puede especificar qué columnas se quieren, qué vistas han de mostrar estas columnas y el fichero XML que define la apariencia de las mismas. En el caso que se está viendo no todos los items han de ser tratados por igual (no todos contienen imágenes, por ejemplo), por lo que se ha optado por hacer una implementación específica de este adaptador. A continuación se detallan aspectos importantes de la clase:

```
1  public CursorPodcastListAdapter(Context context, int layout,
       Cursor c,
2                                  String[] from, int[] to) {
3      super(context, layout, c, from, to);
4      this.context = context;
5      this.layout = layout;
6      this.inflater = LayoutInflater.from(context);
7  }
```

Constructor de la clase. En él se obtiene una instancia de un objeto de tipo `LayoutInflater`, la cuál se utilizará más tarde para instanciar los objetos de tipo `View` a partir del fichero XML. Recibe parámetros muy importantes:

- layout: el fichero que define la apariencia de cada item.
- c: `Cursor` con los datos que conforman el listado de items.
- from: `array` con las columnas que se quieren mostrar.
- to: `array` con las vistas que han de mostrar las columnas.

```
1  public View newView(Context context, Cursor cursor, ViewGroup
       parent) {
2      ViewHolder holder = new ViewHolder();
3      View row = inflater.inflate(layout, parent, false);
4
5      holder.title = (TextView) row.findViewById(R.id.
           list_podcasts_row_title);
6      holder.last_checked =(TextView)row.findViewById(R.id.
           list_podcasts_row_last_checked);
7      holder.image = (ImageView)row.findViewById(R.id.
           podcast_image);
8
```

```
9       row.setTag(holder);
10      return row;
11  }
```

Este método es el encargado de "inflar" las vistas a partir del fichero XML. Para ello hace uso del objeto `LayoutInflater` y su método `inflate()`, obtenido en el constructor de la clase.

```
1   public void bindView(View convertView, Context context,
        Cursor c) {
2       ViewHolder holder;
3       holder = (ViewHolder) convertView.getTag();
4       int titleCol = c.getColumnIndex(DroidCatcherDB.
            KEY_PODCAST_TITLE);
5       String sTitle = c.getString(titleCol);
6       int checkedCol = c.getColumnIndex(DroidCatcherDB.
            KEY_PODCAST_LAST_CHECKED);
7       String sChecked = c.getString(checkedCol);
8       holder.title.setText(sTitle);
9       holder.last_checked.setText(sChecked);
10      File podcastFile = StorageUtils.getPodcastDir(context,
            sTitle);
11      Bitmap img = BitmapFactory.decodeFile(podcastFile.
            toString()+"/"+"image.jpg");
12      if(img != null){
13                  holder.image.setImageBitmap(img);
14      }else{
15                  holder.image.setImageResource(R.drawable.
                        podcast);
16      }
17  }
```

Se encarga de mapear los datos de las columnas pasadas en el constructor de la clase en el parámetro `from`, con sus correspondientes vistas pasadas en el parámetro `to`

Como se puede ver, en ambos métodos se hace uso de una clase interna especial llamada `ViewHolder`. Esta clase, que contiene referencias a las vistas sobre las cuales se van a mapear los datos, es almacenada en el atributo "tag" del objeto `View` creado en `newView()`, de modo que cada vez que se llame al método `bindView()` con nuevos datos que mapear, no sea necesario crear nuevas instancias de las vistas mediante `findViewById`.

4.3.7. Otros elementos importantes de la interfaz de usuario

En la sección anterior se ha explicado detalladamente el funcionamiento y la creación de la interfaz de usuario principal, así como el de una Activity importante como es el listado de podcasts. El resto de Activities y elementos de la interfaz de usuario funcionan de manera muy similar, por lo que no hace falta entrar en mayores explicaciones acerca de su desarrollo. Sí existen, sin embargo, algunos aspectos peculiares como el uso de elementos visuales comunes para diversas Activities y que se explicarán en este apartado.

4.3.8. Layouts comunes

Dentro de la aplicación, existen algunos elementos de la interfaz de usuario que son comunes a todas las Activities. Es el caso, por ejemplo, de *now_playing_layout.xml*. Este layout es mostrado en pantalla cuando se está reproduciendo cualquier episodio y es un "acceso directo" a la pantalla de reproducción de la aplicación. A continuación se muestra cómo es este elemento en sí y cómo es integrado en todas las actividades de la aplicación.

now_playing_layout.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/
    android">
<com.google.android.droidcatcher.view.RelativePanel
        android:id="@+id/now_playing_panel"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:paddingLeft="5px"
        android:paddingTop="5px"
        android:paddingRight="5px"
        android:paddingBottom="5px"
        android:gravity="bottom"
        android:layout_gravity="bottom"
        android:visibility="gone"
        android:layout_alignParentBottom="true">

        <ImageButton android:id="@+id/button_go_to_player"
            android:layout_width="24px"
            android:layout_height="24px"
            android:background="@android:drawable/
               ic_media_play"
            android:onClick="clickNowPlayingHandler"
            android:gravity="right"
            android:layout_alignParentRight="true"/>
```

```
23  </com.google.android.droidcatcher.view.RelativePanel>
24  </merge>
```

Se trata de un componente bastante sencillo. Se hace uso de la clase `RelativePanel`, que extiende de la clase `RelativeLayout` pero personalizada con colores y dimensiones para la ocasión. Dentro de este elemento solamente se hace uso de un botón. En este caso, la etiqueta `merge` se utiliza en conjunción con la etiqueta `include` (que ahora se verá) y que se utiliza para simplificar el número de instancias de la clase `View` que Android genera automáticamente a partir de los ficheros de layouts.

Para incluir en una Activity cualquiera este elemento, basta con añadir en su layout la siguiente línea:

```
1  <include layout = "@layout/now_playing_layout" />
```

Como se verá en el apartado de servicios, todas las Activities tienen que estar conectadas de alguna forma con los servicios que ofrece la aplicación, especialmente con el servicio `AudioPlayerService`.

4.4. Construcción y acceso a la base de datos

`DroidCatcherDB` es la clase principal para el manejo de la BBDD en DroidCatcher. Se encuentra dentro del paquete **com.google.android.droidcatcher.db**. Como se ha visto en anteriores capítulos, la BBDD utilizada es **SQLite3** [56].

A continuación, se van a explicar los aspectos más importantes de esta clase así como del uso que el resto de Activities hacen de ella.

La clase utiliza los objetos `DataBaseHelper`(`mOpenHelper`) y `SQLiteDatabase`(`db`) para gestionar el acceso a BBDD:

- **DataBaseHelper**: esta clase ayuda a abrir, crear y actualizar el fichero donde se almacena la base de datos.
- **SQLiteDatabase**: expone métodos para gestionar la base de datos SQLite. Esta clase tiene métodos para crear, borrar, ejecutar comandos SQL y realizar otras tareas comunes para la gestión de la base de datos.

4.4.1. DataBaseHelper

Es una clase interna que extiende a la clase `SQLiteOpenHelper`. Hay que crear la subclase mencionada para implementar los métodos `onCreate()`,

`onUpgrade()` y, opcionalmente, `onOpen()`. En este caso, solo se han implementado los métodos obligatorios, teniendo especial importancia el método `onCreate()`.

```
1  public void onCreate(SQLiteDatabase db) {
2     try{
3               db.execSQL(CREATE_TABLE_PODCAST);
4               db.execSQL(CREATE_TABLE_EPISODE);
5               db.execSQL(CREATE_TABLE_CATEGORY);
6
7               db.execSQL("insert into category (name, icon)
                   values (\"Art\", \"art_icon\")");
8               db.execSQL("insert into category (name, icon)
                   values (\"Books\", \"books_icon\")");
9               db.execSQL("insert into category (name, icon)
                   values (\"Business\", \"business_icon\")");
10              db.execSQL("insert into category (name, icon)
                   values (\"Comedy\", \"comedy_icon\")");;
11          ...
12          ...
13          ...
14              }
15              catch (Exception e)
16              {
17                 Log.i(DATABASE_NAME,"tables already exist");
18              }
19          }
```

Este método es llamado cuando se crea la base de datos por primera vez. Aquí es donde se crean las tablas para podcast, episodios y categorías y donde, además, se rellena la tabla de categorías disponibles en la aplicación. Recibe por parámetro un objeto de tipo `SQLiteDatabase` que, como se ha visto antes, permite realizar las tareas más comunes en la gestión de una base de datos (creación, borrado, ejecución de comandos SQL, etc).

4.4.2. SQLiteDatabase

Este objeto es el que se utiliza para realizar todas las consultas y actualizaciones necesarias en las tablas de la base de datos. Se obtiene a partir del objeto visto anteriormente, `DataBaseHelper`, y normalmente se obtendrá de dos formas distintas, en función de qué tipo de consultas o sentencias SQL se quieran ejecutar:

- Para los **métodos de búsqueda** funciona de la siguiente manera: Se crea un objeto `SQLiteDatabase` mediante el método

`mOpenHeler.getReadableDatabase()`. Una vez que se tiene este objeto se llama al método `db.query()`, el cuál devolverá un `Cursor` con el resultado de la consulta.

- Para **insertar nuevos registros** en BBDD: Al igual que en el caso de la lectura, se ha de obtener una instancia de `SQLiteDatabase(db)` pero esta vez tiene que tratarse de una instancia con permisos de escritura. Esto se consigue mediante la llamada al método `mOpenHelper.getWritableDatabase()`. Una vez conseguida dicha instancia se llamará al método `db.insert()` para insertar nuevos registros en BBDD. A este método hay que pasarle por parámetro un objeto de tipo `ContentValues`, que se habrá creado previamente, y que contendrá los valores de las columnas que se quieran rellenar.

- Para **borrar registros** de BBDD se procede de forma muy similar a la inserción. Se ha de obtener también en este caso una instancia de `SQLiteDatabase` con permisos de escritura mediante la llamada al método `mOpenHelper.getWritableDatabase()`. Una vez obtenida esta instancia se hace uso del método `db.delete()` para borrar un registro determinado de BBDD.

A continuación se muestran ejemplos para los casos citados:

Consulta:

```
1  public Cursor fetchPodcast(long podcastId) throws
       SQLException {
2          db = mOpenHelper.getReadableDatabase();
3          Cursor result = db.query(true, PODCAST_TABLE, new
               String[] {
4                    KEY_PODCAST_ID, KEY_PODCAST_TITLE,
                         KEY_PODCAST_DESCRIPTION,
                         KEY_PODCAST_CATEGORY_ID,
5          KEY_PODCAST_URL,KEY_PODCAST_IMAGE_URL,
               KEY_PODCAST_LAST_CHECKED,
               KEY_PODCAST_LATEST_EPISODE_ID,
6          KEY_PODCAST_LATEST_EPISODE_TITLE}
7          , KEY_PODCAST_ID + "=" + podcastId, null, null, null,
               null, null);
8          return result;
9       }
```

Inserción:

```
1  public long insertPodcast(String title, String description,
       long categoryId, String url, String image_url) {
```

```
        db = mOpenHelper.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(KEY_PODCAST_TITLE, title);
        values.put(KEY_PODCAST_DESCRIPTION, description);
        values.put(KEY_PODCAST_CATEGORY_ID, categoryId);
        values.put(KEY_PODCAST_URL, url.toString());
        values.put(KEY_PODCAST_IMAGE_URL, image_url != null ?
            image_url.toString() : "");
        long result = db.insert(PODCAST_TABLE, null, values);
        db.close();
        return result;
     }
```

Borrado:

```
public boolean deletePodcast(Long podcastId) {
        db = mOpenHelper.getWritableDatabase();
        boolean result = (db.delete(PODCAST_TABLE,
           KEY_PODCAST_ID + "=" + podcastId, null) > 0);
        db.close();
        return result;
    }
```

La utilización de cualquiera de estos métodos desde una Activity es muy sencilla. A continuación se muestra un ejemplo de código en el que se hace una consulta a base de datos desde una Activity, obteniendo como resultado un objeto de tipo `Cursor`:

```
    DroidCatcher mdb = new DroidCatcherDB(this);
    Cursor c = mdb.fetchPodcast(podcastId);
        startManagingCursor(c);
        c.moveToFirst();
        String title = c.getString(c.getColumnIndex(
           DroidCatcherDB.KEY_PODCAST_TITLE));
        String url = c.getString(c.getColumnIndex(
           DroidCatcherDB.KEY_PODCAST_URL));
        String description = c.getString(c.getColumnIndex(
           DroidCatcherDB.KEY_PODCAST_DESCRIPTION));
        c.close();
```

4.5. Servicios: Reproducción de episodios, servicio de descarga, servicio de búsqueda de actualizaciones

DroidCatcher consta de tres servicios fundamentales para el buen funcionamiento de la aplicación. Los dos primeros servicios, la reproducción y la descarga, ofrecen la posibilidad al usuario de interactuar con ellos, mientras que el tercero de ellos se trata de un servicio independiente que realizará una determinada tarea y se terminará después de ello.

4.5.1. Gestión de los Servicios

Existe una clase fundamental dentro de DroidCatcher para la gestión y el manejo de los servicios durante la ejecución de la aplicación. Esta clase es `ServiceUtils`. Esta clase, localizada dentro del paquete `com.google.android.droidcatcher.service`, podría considerarse como una clase de utilidades que contiene métodos para la creación de un servicio, el borrado del servicio, así como de los métodos que permiten establecer la asociación entre el servicio iniciado y la aplicación. Esta clase es común para los tres servicios de los que consta DroidCatcher, por lo que siempre habrá que hacer uso de ella para iniciar cualquier servicio de la aplicación.

A continuación se explicarán los métodos más relevantes para uno de los servicios, concretamente el servicio de descargas. Para el resto de servicios, los métodos utilizados serán muy similares por lo que su explicación resultaría muy redundante.

Lo primero que hay que destacar de esta clase, son los atributos que tiene:

```
1 public static IDownloaderService sDownloaderService = null;
2 private static HashMap<Context, DownloaderServiceBinder>
    sDownloadConnectionMap = new HashMap<Context,
    DownloaderServiceBinder>();
```

El primero de ellos no necesita mucho que comentar. Se trata de la interfaz que representa al servicio. El segundo atributo visto arriba es un objeto `HashMap`, donde se almacenará quienes han establecido una unión con el servicio de descarga. Es decir, cada vez que alguien quiera hacer uso del servicio, quedará "registrado" en este objeto. Este registro, mostrado a continuación, se realiza en el método `bindToDownloaderService()` y es clave para saber cuándo el servicio no es necesario que siga ejecutándose y, por tanto, podrá ser terminado:

```
1  public static boolean bindToDonwloaderService(Context context
       , ServiceConnection callback) {
2          context.startService(new Intent(context,
               DownloaderService.class));
3          DownloaderServiceBinder sb = new
               DownloaderServiceBinder(callback);
4          sDownloadConnectionMap.put(context, sb);
5          return context.bindService((new Intent()).setClass(
               context,DownloaderService.class), sb, Service.
               BIND_AUTO_CREATE);
6  }
```

En la primera línea de código, la llamada a `startService()` se encarga de crear el servicio. Nótese que múltiples llamadas no resultan en la creación de múltiples servicios: si el servicio ya se estaba ejecutando, entonces seguirá así. Si no, entonces se instanciará y empezará a ejecutarse.
`DownloaderServiceBinder` es la clase que implementa la interfaz `ServiceConnection`, por lo que es necesario obtener una instancia de ella.
Como se ha mencionado más arriba, esta instancia del objeto `DownloaderServiceBinder` es almacenada en el `HashMap`. Finalmente, se hace la llamada a `context.bindService()`, de modo que se establezca la unión entre el servicio y aquél que lo haya invocado, pudiendo interactuar entre ambos.

El siguiente método a destacar es `unbindFromDownloaderService()`, que se podría considerar como el opuesto al explicado en el apartado anterior:

```
1  public static void unbindFromDownloaderService(Context
       context) {
2          DownloaderServiceBinder sb = (DownloaderServiceBinder
               ) sDownloadConnectionMap.remove(context);
3          if (sb == null) {
4              Log.e("ServiceUtils", "Trying to unbind for
                   unknown Context");
5              return;
6          }
7          context.unbindService(sb);
8          if (sDownloadConnectionMap.isEmpty()) {
9              // presumably there is nobody interested in the
                   service at this point,
10             // so don't hang on to the ServiceConnection
11             sDownloaderService = null;
12         }
13 }
```

Como el lector se puede imaginar por el nombre del método, éste es el

encargado de romper el lazo de unión entre el servicio y aquél que lo haya invocado. Para ello, primeramente se borra esta unión del `HashMap` donde se guardó en la creación del servicio. Una vez borrado este "registro", se procede a la llamada del método `context.unbindService()`, que es dónde se hace efectiva la desunión del servicio con el contexto. Finalmente, se comprueba si queda alguien que esté utilizando el servicio para, en caso contrario, eliminar la instancia de `ServiceConnection`.

4.5.2. Servicio de Audio

El servicio de audio, `AudioPlayerService` dentro del código, es el encargado de la reproducción de los episodios y de que ésta no pare aunque se salga de la aplicación (siempre y cuando el proceso principal de la aplicación siga en ejecución).

Como se ha comentado antes, es un servicio que ofrece métodos para interactuar con la propia aplicación y, por ende, con el usuario final de DroidCatcher. A continuación, se explicará de forma detallada el proceso de creación del servicio, así como de la interfaz remota necesario para la comunicación entre la interfaz de usuario y el propio servicio.

Para crear este tipo de servicios hacen falta tres pasos:

1. **Definir interfaz AIDL**: interfaz donde se definen los métodos que el servicio expondrá al resto de componentes de la aplicación.
2. **Crear la clase Servicio**: es la clase que extiende de `Service` y que representa al servicio en sí mismo.
3. **Añadir el servicio en AndroidManifest.xml**: es necesario añadir el servicio en el fichero de manifiesto de la aplicación.

4.5.3. Interfaz AIDL

En el capítulo *2.- Estado del Arte*, se explicó en detalle el funcionamiento de estas interfaces así como el lenguaje AIDL [31] por lo que simplemente se verá la interfaz diseñada para el servicio de audio de la aplicación. **IAudioPlayerService.aidl**:

```
1 package com.google.android.droidcatcher.service;
2
3 interface IAudioPlayerService{
4     long getCurrentPosition();
5     long getDuration();
6     boolean streamTrack(in long id);
```

```
7       String getTitle ();
8       long getId ();
9       void setId (long id);
10      void pauseTrack();
11      void startTrack();
12      void seekTrack(long progress);
13      void stopTrack();
14      boolean isPlaying();
15      boolean isActive();
16      boolean isRemote();
17      String getError();
18      void openFile(String path);
19      void openFileAsync(String path);
20  }
```

Como se puede ver, los métodos definidos en la interfaz son los propios de un reproductor de audio, además de algún método para mostrar datos de interés al usuario final de la aplicación.

Una vez definido la interfaz AIDL, se pasa a crear la propia clase que representa el servicio (extiende la clase Service), `AudioPlayerService`. Hay varios aspectos a destacar dentro de la clase, que a continuación se detallan:

- La clase extiende a `Service`. Para tener control básico de la funcionalidad del servicio se sobrescriben los métodos `onCreate()` y `onDestroy()`.
- Se implementan los métodos definidos en la interfaz AIDL
- Se utiliza una clase interna, `MultiPlayer`, para que sea ella la encargada de lidiar con los archivos multimedia.
- Hay que manejar eventos como llamadas entrantes que afectan a la reproducción de episodios.

En los métodos `onCreate()` y `onDestroy()` se inicializarán y destruirán los objetos más importantes para el funcionamiento del servicio.

onCreate:

```
1   public void onCreate() {
2       super.onCreate();
3       mAudioManager = (AudioManager) getSystemService(Context.
            AUDIO_SERVICE);
4       mp = new MultiPlayer();
5       mp.setHandler(mUpdateHandler);
6   }
```

Este método es el primero que se llama cuando el servicio es creado por primera vez, por lo tanto hay que inicializar objetos que se vayan a utilizar durante toda la vida del servicio. El objeto `MultiPlayer` es el encargado de la reproducción de los archivos multimedia mientras que el objeto `AudioManager` proporciona acceso al control de llamadas y de volumen.

onDestroy():

```
1 public void onDestroy() {
2     mp.stop();
3     mp.release();
4     mp = null;
5     mAudioManager.abandonAudioFocus(mAudioFocusListener);
6     super.onDestroy();
7 }
```

este método es llamado por el sistema para notificar al servicio que ya no está siendo usado y que, por tanto, va a ser destruido. El servicio debería liberar recursos en este punto. Una vez que se vuelva de este método ya no habrá más llamadas al servicio y estará definitivamente muerto. En este caso, se liberan los objetos `Multiplayer` y se retira el foco de audio.

Android proporciona, dentro de las librerías disponibles para el programador, las clases necesarias para gestionar la reproducción de ficheros multimedia. La clase `MediaPlayer` puede ser utilizada para controlar la reproducción tanto de audio como de vídeo. Esta reproducción puede ser de ficheros almacenados en el sistema de archivos, o bien de flujos de datos obtenidos de la red.

En DroidCatcher, además de la clase `MediaPlayer`, se ha hecho uso de otra clase proporcionada por Android, `AudioManager`. Esta clase proporciona los métodos necesarios para la reproducción en sí misma (play, stop, pause, next, previous). Además de éstos, contiene métodos de gran utilidad que ofrecen información útil para la interfaz de usuario. A continuación se detallan algunos de los métodos comentados:

- isPlaying(): devuelve true si se está reproduciendo un capítulo en ese momento
- getDuration(): devuelve la duración del episodio reproducido
- getTitle(): devuelve el título del episodio reproducido
- getId(): devuelve el identificador del episodio reproducido
- seek(): avanza hasta una posición de la reproducción
- setVolume(): establece el volumen

Acceso a fichero Multimedia

El acceso al archivo físico del episodio se hace de forma relativamente sencilla. Dentro de la clase interna `MultiPlayer` se han creado dos métodos para tal propósito, uno de ellos de forma síncrona y el otro de forma asíncrona. Dichos métodos no tienen más que indicarle al reproductor de Android donde está la fuente de datos y él se encargará de hacer el resto del trabajo.

4.5.4. Servicio de Descargas

Es el servicio encargado de realizar las descargas de los episodios de cada podcast. La clase principal es `DownloaderService`.

Limitaciones iniciales: solamente se permite una descarga simultánea de episodios. El resto de capítulos que sean seleccionados para descargar mientras se está produciendo una descarga, serán encolados, pasando a procesarse una vez haya terminado el anterior proceso de descarga. Se ha tomado esta decisión por cuestiones prácticas y de rendimiento. Varias descargas simultáneas podrían provocar un bajón de rendimiento y un mayor consumo de batería del teléfono. Además, dado que la aplicación contiene su propio sistema de descargas automáticas y actualizaciones, podría suceder que sea el propio sistema el que intente descargarse demasiados capítulos al mismo tiempo sin el propio consentimiento del usuario.

Dentro del servicio existe un objeto `Thread`, `DownloaderThread`, que es el encargado de realizar la descarga en sí. Para ello solamente necesita la URL del capítulo a descargar y un lugar donde almacenar el capítulo en el sistema de ficheros de Android.

Básicamente, la implementación del servicio de descargas es muy similar a la del servicio de reproducción de episodios vista en el apartado anterior. A continuación se explicarán los aspectos más destacables de este servicio, y que lo diferencian del servicio de reproducción de episodios.

La descarga en sí misma de los capítulos la realiza un hilo creado dentro de la propia clase del servicio y llamado `DownloaderThread`. A continuación se muestra la llamada a dicho hilo:

```
1 private boolean startDownload() throws DownloaderException {
2     //If mCurrentEpisode isn't null we're already downloading
          something...
3     if (mCurrentEpisode != null) return false;
4     //No more queued downloads?
5     if (mDownloads.size() < 1) return false;
6     DownloaderThread downloaderThread= new DownloaderThread(
          this);
7     downloaderThread.start();
```

```
8       return true;
9   }
```

`DownloaderThread` es el encargado de realizar la petición HTTP correspondiente y de procesar la respuesta obtenida. A continuación se muestran las partes más relevantes del método `run()`, que son las correspondientes a la creación y procesamiento de la petición HTTP:

```
1       public void run(){
2            ...
3       ...
4       ...
5       mHttpClient = new DefaultHttpClient();
6       HttpGet GET = new HttpGet(mCurrentEpisode.getUrl());
7            BasicHttpResponse response = (BasicHttpResponse)
                mHttpClient.execute(GET);
8            if (response.getStatusLine().getStatusCode() == 401)
                 {
9               throw new DownloaderException("Unathorised access
                    - check your username and password.");
10           } else if (response.getStatusLine().getStatusCode()
                  != 200) {
11                   throw new DownloaderException("Error
                        connecting to server.");
12           }
13      ...
14      ...
15      ...
16      FileOutputStream out = new FileOutputStream(f);
17
18      InputStream is = entity.getContent();
19      BufferedInputStream bi = new BufferedInputStream(is);
20
21      byte[] buffer = new byte[10 * 1024]; //Buffer 10K
22      int result;
23      mPercent = 0;
24      long index = 0;
25      long time = System.currentTimeMillis();
26
27      while (true) {
28          if (stopFlag) {
29              stopFlag = false;
30              throw new DownloaderException("Download canceled"
                    );
31          }
32          result = bi.read(buffer);
33          if (result == -1) break;
34          index += result;
```

```
35          if (time < System.currentTimeMillis() + 3000) {
36              time = System.currentTimeMillis();
37              mPercent = (int)(index * 100 / fileSize);
38              notifyChangeProgress(S_DOWNLOADING, mPercent);
39          }
40          out.write(buffer, 0, result);
41
42      }
```

Comunicación entre Servicio e Interfaz de Usuario

La Activity `DownloadsList` muestra al usuario el estado de las descargas actuales. Para ir actualizando esta información necesita establecer una comunicación con el servicio de descargas. Para ello, hace uso de dos `BroadcastReceiver`, uno para el progreso de la descarga (`mProgressListener`) y otro para el estado (`mStatusListener`). Cada uno de estos *receivers* necesitará ser registrado para recibir las notificaciones procedentes del servicio de descarga.

A continuación se muestra el código para realizar estas tareas para uno de los receivers, `mStatusListener`:

creación del receiver:

```
1  private BroadcastReceiver mStatusListener = new
       BroadcastReceiver() {
2           public void onReceive(Context context, Intent intent)
                {
3               String action = intent.getAction();
4               long _id = intent.getLongExtra("id", -1);
5               if (action.equals(DownloaderService.
                    S_START_DOWNLOAD)) {
6                       if(_id != -1){
7                           updateDownloadsList();
8                       }
9               }if (action.equals(DownloaderService.
                    S_DOWNLOAD_QUEUED)) {
10                      if(_id != -1){
11                          updateDownloadsList();
12                      }
13              }else if (action.equals(DownloaderService.
                    S_DOWNLOADED)) {
14                      updateDownloadsList();
15              } else if (action.equals(DownloaderService.
                    S_DOWNLOAD_ERROR)) {
16                      updateDownloadsList();
```

```
17              }else if (action.equals(DownloaderService.
                    S_DOWNLOAD_CANCELED)) {
18                  if(_id != -1){
19                          updateDownloadsList();
20                  }
21              }
22          }
23      };
```

Además de crear el objeto `BroadcastReceiver` es necesario implementar el método `onReceive()`, qué donde se recibirán los mensajes procedentes del servicio. Aquí se deberá hacer distinción del tipo de mensaje recibido y actuar en consecuencia.

Registrar el BroadcastReceiver:

```
1   protected void onStart(){
2       super.onStart();
3       if (false == ServiceUtils.bindToDonwloaderService(
            getApplicationContext(), mDownloaderConnection)) {
4               // something went wrong
5               mUpdateHandler.sendEmptyMessage(QUIT);
6       }
7           IntentFilter f = new IntentFilter();
8           f.addAction(DownloaderService.S_DOWNLOADED);
9           f.addAction(DownloaderService.S_DOWNLOAD_QUEUED);
10          f.addAction(DownloaderService.S_START_DOWNLOAD);
11          f.addAction(DownloaderService.S_DOWNLOAD_ERROR);
12          f.addAction(DownloaderService.S_DOWNLOAD_CANCELED);
13          registerReceiver(mStatusListener, new IntentFilter(f)
                );
14  }
```

Importante el objeto `IntentFilter`, mediante el cuál se indica qué acciones tendrá en consideración el `BroadcastReceiver`.

Desde el servicio:

```
1   private void notifyChange(String what, Episode episode) {
2           Intent i = new Intent(what);
3           if(S_START_DOWNLOAD.equals(what) ||
4           S_DOWNLOAD_QUEUED.equals(what) ||
5           S_DOWNLOAD_CANCELED.equals(what) ||
6           S_DOWNLOADED.equals(what)){
7               i.putExtra("id", episode.get_id());
8           }
9           sendBroadcast(i);
10      }
```

Desde el servicio de Descargas se notificará de cualquier cambio que se produzca en el estado de las descargas. Para ello se creará un objeto `Intent` indicando el tipo de mensaje a enviar.

4.5.5. Servicio de Actualizaciones Automáticas

Una de las partes mas interesantes de la aplicación DroidCatcher es la posibilidad que ofrece de configurar un servicio de actualizaciones automáticas de los podcasts a los que está suscrito el usuario. Mediante este servicio, la aplicación comprobará de forma periódica la existencia de nuevos episodios aunque el usuario no tenga la aplicación abierta. El usuario podrá activar y desactivar esta funcionalidad, así como configurar la periodicidad de estas actualizaciones. Dichas actualizaciones automáticas llevarán consigo sus correspondientes notificaciones al usuario. A continuación se explican en detalle los aspectos más relevantes del servicio de actualización, así como del funcionamiento de las actualizaciones al usuario.

UpdateService Al igual que en los casos de los servicios de reproducción y descargas, la clase principal para el servicio de actualizaciones extiende de la clase `Service`. Sin embargo, a diferencia de los anteriores, este servicio no interactúa directamente con el usuario de la aplicación y, por tanto, no es necesario definir la interfaz AIDL con los métodos que se podrán ejecutar desde una Activity. En este caso, además de implementar el método `onCreate()` como en los servicios anteriores, se ha de implementar también el método `onStartCommand()`, que se ejecutará siempre inmediatamente después de `onCreate`.

```
1 public int onStartCommand(Intent intent, int flags, int
    startId) {
2     searchForUpdates();
3     return super.onStartCommand(intent, flags, startId);
4 }
```

Como el lector se puede imaginar, el método `searchForUpdates()` contiene la lógica necesaria para comprobar si existe algún episodio nuevo para los podcasts a los que está suscrito el usuario. Dicha lógica es muy similar a la ya explicada en el apartado de suscripción a podcasts, por lo que no se entrará en detalles otra vez.

Un aspecto que si merece la pena destacar es la inclusión de notificaciones al usuario en caso de encontrar actualizaciones disponibles. Dichas notificaciones se producen dentro del método `notifyUpdates()`, que a continuación se explica en detalle:

```
1  private void notifyUpdates(Podcast podcast){.
2      Notification notif = new Notification();
3      PendingIntent contentIntent = PendingIntent.getActivity(
           this, 0,
4                  new Intent(this, GetMoreEpisodesList.class)
5                          .setFlags(Intent.
                               FLAG_ACTIVITY_NEW_TASK)
6                          .putExtra("podcast", podcast.get_id()
                               ),
7                  PendingIntent.FLAG_UPDATE_CURRENT);
8
9      notif.contentIntent = contentIntent;
10     notif.tickerText = getText(R.string.
           new_episodes_available);
11     notif.icon = R.drawable.notification_icon;
12
13      // our custom view
14         RemoteViews contentView = new RemoteViews(
               getPackageName(), R.layout.status_bar_notification
               );
15         contentView.setTextViewText(R.id.
               episodes_notification_text, podcast.getTitle());
16         contentView.setImageViewResource(R.id.
               episodes_notification_icon, R.drawable.
               notification_icon);
17         notif.contentView = contentView;
18
19         nm.notify(UPDATE_NOTIFY_ID , notif);
20
21     }
```

Para realizar notificaciones al usuario se utiliza el método `notify()` de la clase `NotificationManager` (nm). Dicho método necesita como uno de sus parámetros un objeto de tipo `Notification`. Como se puede ver en el ejemplo, dicho objeto contendrá información importante como el texto a mostrar, un icono y la acción a realizar (`PendingIntent`).

4.6. Procesamiento de archivos XML. Peticiones HTTP y subscripciones RSS

DroidCatcher ofrece al usuario la posibilidad de hacer búsquedas de podcasts desde la misma aplicación y de poder suscribirse a los mismos de manera muy rápida y sencilla. Para ello, DroidCatcher hace uso de dos servicios disponibles en Internet (uno en castellano y otro en inglés) y que están a

disposición de cualquier usuario que los quiera utilizar.

Ambos servicios, **DigitalPodcast** [57] y **Podsonoro** [58], permiten acceder a realizar búsquedas de contenidos desde la mayor parte de lenguajes de programación. Esto significa que se puede integrar un buscador DigitalPodcast y/o Podsonoro fácilmente dentro de una aplicación, en este caso dentro de DroidCatcher. Se trata de servicios basados en REST (Representational State Transfer) y se utilizan mediante peticiones HTTP, del mismo modo que un navegador normal. Las peticiones HTTP se crean usando una URL de entrada (`http://digitalpodcast.com/podcastsearchservice/v1b/` y `http://www.podsonoro.com/searchservice/`) y añadiendo argumentos al final de la misma. Dichos argumentos servirán para establecer los parámetros de búsqueda (palabras clave, número de resultados, formato de salida, etc.). Los resultados devueltos por el servicio tienen formato XML y podrá ser en concreto RSS, OPML o bien OPML RSS. La aplicación por tanto realizará una petición HTTP, obtendrá una respuesta en uno de estos formatos y extraerá la información interesante de la misma.

DigitalPodcast impone dos requerimientos técnicos para el uso de su servicio:

- Requiere que la aplicación que haga uso del servicio esté registrada en su página web y, por tanto, tenga asignado un ID. Este ID debe ser enviado en cada petición (parámetro appid).
- La frecuencia: con la que una máquina puede hacer peticiones al servicio está limitada. Actualmente está limitación es de 1000 peticiones al día.

La clase `DigitalPodcastSearch` es la encargada de obtener el listado de resultados para una búsqueda determinada mediante el método `searchPodcast()`. Se realiza una petición HTTP con el formato definido en DigitalPodcast.com. La respuesta a dicha petición es un flujo de datos en formato XML que deben ser procesados con un *parser*, en este caso, se utiliza **SAX**. Para ello, se ha creado un Handler propio para este tipo de parsers, `SearchPodcastHandler`, que extiende a `DefaultHandler`, y que implementa los métodos necesarios para parsear la información contenida en el flujo de datos XML. Estos métodos son:

- **startElement()**: este método es llamado cuando se empieza a procesar un nuevo elemento.
- **endElement()**: este método es llamado cuando se termina de procesar un elemento.
- **characters()**: método encargado de procesar los datos contenidos dentro de un elemento.

A continuación se muestra cómo realizar las peticiones HTTP

```
if(searchEngine!=null && searchEngine.equalsIgnoreCase("
    DIGITALPODCAST"))
            url =DIGITAL_PODCAST_SEARCH_URL +keywords.replace
                (" ","%20");
        else
            url = PODSONORO_SEARCH_URL +keywords.replace(" ",
                "%20");

        HttpGet get = new HttpGet(url);
        ResponseHandler<String> responseHandler = new
            BasicResponseHandler();
        String mytext;
        try {
                mytext = httpClient.execute(get,
                    responseHandler);
        } catch (Exception e) {
                throw new DownloaderException("Could not
                    download from url '" + url + "'");
        }
     ...
     ...
     ...
```

El código de más abajo muestra cómo se crean los objetos necesarios para procesar la respuesta XML obtenida:

```
SearchPodcastHandler searchHandler = new SearchPodcastHandler
    ();
        SAXParserFactory saxParserFactory = SAXParserFactory.
            newInstance();
        saxParserFactory.setValidating(false);
        SAXParser saxParser;
        try {
                saxParser = saxParserFactory.newSAXParser();
        } catch (Exception e) {
                throw new RuntimeException("Could not create
                    XML parser", e);
        }
        ByteArrayInputStream textBytes = new
            ByteArrayInputStream(mytext.getBytes());
        try {
                saxParser.parse(textBytes, searchHandler);
        } catch (Exception e) {
                throw new RuntimeException("Failed to parse
                    feed", e);
        }
```

```
16
17          ArrayList<Podcast> podcasts = searchHandler.
               getPodcasts();
```

4.7. Preferencias de DroidCatcher

DroidCatcher utiliza un fichero de preferencias para almacenar los parámetros de configuración básica de la aplicación. Para ello hace uso de la clase `SharedPreferences` de Android, con la cuál se puede acceder a modificar estas preferencias de forma rápida y sencilla. A continuación se explica tanto el proceso de definir y crear estas preferencias para que sean mostradas al usuario mediante un sencillo interfaz gráfico, así como el proceso de modificar o acceder a estas preferencias durante la ejecución de la aplicación.

4.7.1. Definición de las preferencias en XML

Al igual que con la interfaz de usuario, Android ofrece la posibilidad de definir las preferencias y cómo serán presentadas al usuario mediante un fichero XML. Cada una de las clases pertenecientes al paquete *android.preference* podrá ser utilizada para componer las preferencias de la aplicación, así como de aquellas clases que se creen nuevas y que extiendan a una de las anteriores. Con estas clases se podrán crear preferencias tales como campos de texto, casillas para marcar, diálogos, etc. y también podrán ser agrupadas en categorías.

En el caso de DroidCatcher se han creado dos grupos de preferencias, una para los nuevos episodios y otra para la configuración de las notificaciones. A continuación se explican los aspectos más relevantes de este fichero **preferences.xml**

```
1  <PreferenceCategory android:title="@string/
      pref_notifications_title">
2          <CheckBoxPreference
3                  android:key="notifications"
4                  android:title="@string/pref_notifications"
5                  android:summary="@string/
                      pref_notifications_summary"/>
6       <ListPreference
7              android:title="@string/
                  pref_scheduled_notifications"
8              android:summary="@string/
                  pref_scheduled_notifications_summary"
9              android:key="list_scheduled_notifications"
```

```
10              android:entries="@array/scheduled_notifications"
11              android:entryValues="@array/
                   scheduled_notifications_values"
12              android:dependency="notifications"/>
```

En este ejemplo se puede ver como se crean dos preferencias pertenecientes a una misma categoría mediante el uso de `PreferenceCategory`. La primera de ellas es un objeto `CheckBox` y la segunda un listado con varias opciones de donde elegir. El atributo *key* es obligatorio ya que es el identificador único del atributo y el cuál habrá que utilizar para acceder al valor desde el resto de componentes del código. Del resto de atributos, cabe destacar **android:dependency**. Este atributo hace referencia a otra preferencia de la que dependerá, es decir, que si la preferencia de la que depende está desactivada entonces ésta preferencia también lo estará. En este caso, si el `CheckBox` de notificaciones está desactivado, el listado con las notificaciones programadas también lo estará.

Preferencia personalizada:

```
1  <com.google.android.droidcatcher.preference.
       TimePickerPreference
2       android:key="time_to_start_searching"
3       android:title="@string/pref_time_to_start_searching_title
           "
4       android:summary="@string/pref_time_to_start_searching"
5       android:dependency="notifications"
6  />
```

Antes se comentó que dentro del archivo de preferencias se podrían utilizar todas las clases pertenecientes al paquete `android.preference` además de clases creadas por el desarrollador y que extendieran de cualquiera de ellas. Éste es el caso del ejemplo mostrado arriba en el que se hace uso de la clase `TimePickerPreference`, clase personalizada que extiende de `DialogPreference`. La manera de usar esta clase personalizada es exactamente igual que con el resto, teniendo los mismo atributos obligatorios (*key*) y los demás atributos del resto de preferencias.

Los aspectos más importantes a tener en cuenta a la hora de crear una preferencia personalizada son dos: por una parte, se ha de tener claro qué tipo de preferencia se quiere (un listado, un diálogo, un checkbox, etc.) para que la preferencia nueva extienda de la clase adecuada; el segundo aspecto a tener en cuenta es que se han de guardar los datos obtenidos en el momento adecuado.

En este caso se precisaba de una preferencia que fuese un diálogo para seleccionar una hora, por lo que la clase nueva creada extenderá a

`DialogPreference` y hará uso del widget `TimePicker`. Como es una clase personalizada con su propia apariencia visual, se tendrá que sobreescribir el método `onCreateDialogView()` para crear en él el contenido visual: **TimePikerPreference:**

```
protected View onCreateDialogView() {
    TimePicker tp = new TimePicker(getContext());
    tp.setIs24HourView(true);
    tp.setOnTimeChangedListener(this);
    int h = getHour();
    int m = getMinute();
    if (h >= 0 && m >= 0) {
        tp.setCurrentHour(h);
        tp.setCurrentMinute(m);
    }
    return tp;
}
```

A la hora de guardar la preferencia es importante el cuándo. No es lo mismo un `Checkbox`, en el que perfectamente se podría guardar al marcar o desmarcar la casilla, que un diálogo para seleccionar una hora, donde lo más adecuado es guardar la preferencia al cerrar el diálogo. Para ello, se ha sobrescrito el método `onDialogClosed()`:

```
public void onDialogClosed(boolean positiveResult) {
    if( positiveResult ) {
        if( isPersistent() )
            persistString(mHour + ":" + mMinute);
        callChangeListener(mHour + ":" + mMinute);
    }
}
```

De este modo, cada vez que se vayan cambiando los valores de horas y minutos se irán actualizando los atributos de la clase `mHour` y `mMinute`. Hasta que no se cierre el diálogo con el cambio de hora no se guardará en el sistema el nuevo valor de la preferencia.

4.7.2. Consulta de las preferencias desde otros componentes de la aplicación

El acceso a estas preferencias se hace de la misma manera tanto si se hace desde un servicio o desde una actividad. Basta con recuperar una instancia de la clase `SharedPreferences` y a continuación consultar el valor deseado utilizando para ello el atributo `key` de la preferencia en cuestión:

```
1 SharedPreferences settings = context.getSharedPreferences(
     EditPreferences.PREFS_NAME, Context.MODE_PRIVATE);
2 boolean notifications = settings.getBoolean("notifications",
     false);
```

El segundo parámetro del método `getSharedPreferences` determina el modo de operación sobre el fichero de preferencias. Existen tres posibilidades:

- **MODE_PRIVATE**: Es el modo por defecto. Solamente pueden acceder al fichero creado la aplicación que realiza la llamada o las aplicaciones que compartan el mismo identificador de usuario.
- **MODE_WORLD_READABLE**: en este modo todas las aplicaciones tienen acceso al fichero creado para su lectura.
- **MODE_WORLD_WRITEABLE**: todas las aplicaciones tienen acceso al fichero creado en modo escritura.

4.8. Efectos gráficos. Animaciones

Para que DroidCatcher resulte una aplicación más atractiva visualmente hablando se ha hecho uso de algunos efectos gráficos que ofrece Android. Entre estos efectos gráficos se encuentran las **animaciones**. Una animación consiste en aplicar un efecto visual sobre alguno de los componentes que forman la interfaz de usuario de la aplicación [59]. Android ofrece dos tipos de animaciones:

- **Tween Animations**: las animaciones de este tipo pueden realizar transformaciones simples (de tamaño, posición, rotación y transparencia) sobre cualquiera de los contenidos de un objeto `View`. Por ejemplo, sobre un objeto `TextView` se podría mover, rotar o encoger el texto. Si tiene una imagen de fondo, entonces la imagen se transformará junto con el texto. El paquete de Android `android.view.animation` proporciona todas las clases utilizadas para realizar una animación de este tipo.
- **Frame Animations**: este es un tipo de animación más tradicional en el sentido de que es creada con una secuencia de imágenes diferentes, mostradas en orden, como si de un rollo de película se tratase. La clase `AnimationDrawable` es la base para este tipo de animaciones.

Para el caso de DroidCatcher, solamente se han utilizado las animaciones de tipo *tween*, por ser bastante fáciles de utilizar y ofrecer una amplia gama de animaciones predefinidas. A continuación se explica cómo hacer uso de dichas animaciones predefinidas, cómo usarlas sobre un componente y también cómo hacer animaciones propias.

4.8.1. Definición de animaciones

Al igual que con la interfaz gráfica, Android ofrece la posibilidad de definir las animaciones en ficheros XML. Esta forma es la más sencilla y "legible" y, por tanto, es la recomendada frente a definir las animaciones utilizando código Java. Dentro del fichero XML se encuentran las instrucciones de animación que definen la transformación que ha de ocurrir, cuándo ha de ocurrir y cuánto tiempo durará dicha transformación.

A continuación se muestra el código de una animación de Android definida en un fichero XML, que traslada un objeto de la interfaz de usuario desde la parte de arriba de la pantalla hasta la parte de abajo, para finalmente hacerlo desaparecer. El fichero ha de encontrarse en la ruta: `res/anim/`

sllide_view_to_bottom:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <set xmlns:android="http://schemas.android.com/apk/res/
    android"
3     android:interpolator="@android:anim/
        accelerate_interpolator">
4     <translate android:fromYDelta="0%" android:toYDelta="100%
        " android:toXDelta="0" android:duration="1000" />
5     <alpha android:fromAlpha="0.0" android:toAlpha="1.0"
        android:duration="200" />
6 </set>
```

Examinando el fichero en profundidad, se puede observar que se están definiendo 2 animaciónes dentro del fichero: una, definida en el elemento `translate`, que cambiará el componente de posición y la otra, definida en el elemento `alpha`, que modificará la transparencia. Los atributos `from` y `to` definen los valores iniciales y finales de estos parámetros, mientras que `duration` fija la duración de dicha transformación.

Una vez definida la animación en el fichero XML, se puede hacer uso de ella desde un Activity:

ViewPodcast:

```
1 Animation a = AnimationUtils.loadAnimation(target.getContext
    (), R.anim.slide_view_from_bottom);
```

```
2  a.setAnimationListener(new AnimationListener() {
3          public void onAnimationEnd(Animation animation) {
4              episodesList.setVisibility(View.VISIBLE);
5               descriptionTextView.setMaxLines(4);
6               infoDisplayed = false;
7          }
8          public void onAnimationRepeat(Animation animation) {
9          }
10         public void onAnimationStart(Animation animation) {
11             infoFlipper.setVisibility(View.GONE);
12         }
13 });
14 episodesList.startAnimation(a);
```

En la primera línea de código se carga la animación mediante el método `loadAnimation` de la clase `AnimationUtils`. Se ha de pasar por parámetro el fichero XML donde se ha definido previamente la animación. A esta animación se le añade además un *listener*, de modo que reaccione ante eventos tales como el inicio de la animación, el final o la repetición de la misma. Finalmente, con la llamada al método `startAnimation()`, se iniciará la animación.

Capítulo 5

Pruebas

En este capítulo se explicará el plan de pruebas de la aplicación, el diseño de las mismas así como los resultados obtenidos.

En el caso de DroidCatcher, la aplicación puede dividirse fácilmente en módulos claramente diferenciados y realizar pruebas separadas para cada uno de ellos. De hecho, el proceso de desarrollo de la misma ha consistido en: desarrollo módulo uno - pruebas módulo uno; desarrollo módulo dos - pruebas módulo dos y así sucesivamente.

Una vez desarrollados los módulos de la aplicación y probados de forma individual se realiza la integración de los mismos, realizando a su vez las pruebas de funcionamiento correspondientes.

Como ya se ha comentado anteriormente, la aplicación utiliza la versión **Android 2.2 (API nivel 8)**. Todas las pruebas descritas a continuación se han realizado utilizando el emulador disponible en el entorno de desarrollo de Eclipse, utilizando tanto el nivel 2.2 de la plataforma como el posterior, 2.3, para comprobar la compatibilidad de la aplicación con la misma.

Podcasts

Batería de pruebas orientadas a comprobar las tareas comunes con las subscripciones a podcasts. Componen, junto con la reproducción de los capítulos, la funcionalidad más importante de la aplicación, por lo que el funcionamiento ha de ser lo mejor posible.

- **Suscripción a podcasts mediante URL**. Se realiza satisfactoriamente la suscripción introduciendo manualmente la URL de diversos podcasts.
- **Utilización servicio de búsqueda**. Se prueba a buscar suscripciones con diversas palabras clave, tanto en inglés como en español. Se obtie-

nen y muestran al usuario correctamente los resultados de la búsqueda. La suscripción al podcast seleccionado se realiza correctamente.

- **Categorización**. Se muestra correctamente la categoría de los podcasts suscritos. Cambio de categoría de podcast. Ordenación correcta de podcasts en función de la categoría.

- **Borrado de capítulos y podcasts**. Se procede al borrado de capítulos individuales de podcast, así como de podcasts. Se comprueba en el sistema de ficheros que efectivamente se ha realizado el borrado físico de los ficheros de audio e imágenes.

- **Importación capítulos**. Se prueba a importar episodios que han sido descargados por otros medios ajenos a la aplicación.

Servicio de descarga

El servicio de descarga de capítulos también es esencial en la aplicación, por lo que se ha realizado un bloque único de pruebas para certificar su correcto funcionamiento.

- **Descarga de capítulos**. La descarga de capítulos se realiza correctamente. Se controla adecuadamente si la aplicación intenta descargar capítulos que ya no están disponibles en la red. Se comprueba correctamente si el dispositivo dispone de conexión a Internet antes de intentar iniciar la descarga.

- **Cancelación descargas**. Las descargas se cancelan correctamente

- **Comprobación cola de descargas**. Al hacer múltiples descargas simultáneas se establece correctamente la cola de descarga, en el orden adecuado. Según van terminando de descargarse capítulos, se procede a la descarga de los siguientes.

- **Pruebas de rendimiento**. Se comprueba que el rendimiento de la aplicación es correcto miestras se está realizando una descarga. Se realiza una navegación estándar por la aplicación.

Servicio de reproducción

Pruebas básicas de reproducción de capítulos. Se comprueba que el funcionamiento es el de un reproductor sencillo y que, además, se puede estar reproduciendo un capítulo mientras se siguen haciendo el resto de tareas con el teléfono.

- **Reproducción capítulos**. Los episodios son reproducidos correctamente.
- **Pausa, detención capítulos en reproducción**. Los controles del reproductor funcionan correctamente, tanto la pausa, reanudación, parado y cambio de capítulo.
- **Interrupciones de la aplicación (llamadas de teléfono)**. Se comprueba la reacción de la reproducción de un capítulo si se produce una llamada de teléfono. La reproducción se pausa durante la llamada y posteriormente se reanuda de forma automática.
- **Navegación por la aplicación**. Se comprueba que la navegación por la aplicación es correcta durante la reproducción de un episodio. El enlace directo al reproductor desde el resto de Activities es correcto.
- **Pruebas de rendimiento**. Se comprueba que el rendimiento de la aplicación es correcto mientras se está realizando una descarga. Se realiza una navegación estándar por la aplicación.

Servicio de Actualizaciones

- **Configuración de actualizaciones**. Funcionamiento correcto de la configuración del servicio. Los diversos parámetros se actualizan correctamente y el servicio funciona de acuerdo a éstos.
- **Notificaciones**. Se crean las notificaciones en el dispositivo de forma correcta.
- **Comprobación descargas**. Se comprueba que el servicio descarga los capítulos de los podcasts que estén configurados para ello.
- **Pruebas de rendimiento**. Se comprueba que el rendimiento de la aplicación es correcto miestras se está realizando una descarga. Se realiza una navegación estándar por la aplicación.

Pruebas generales de funcionamiento

Una vez hechas las pruebas de cada módulo y realizada la integración correspondiente, se procede a la realización de pruebas generales de toda la aplicación. La finalidad es asegurarse de que todas las funcionalidades siguen intactas después de la integración.

Navegación general por la aplicación, suscribirse a podcasts mediante la URL, buscar podcasts con el servicio integrado en la aplicación y suscribirse

a ellos. Reproducción de capítulos, borrado de los mismos, importar nuevos capítulos, etc.

Capítulo 6

Historia del proyecto

La elección de Android como tema para este proyecto final de carrera tuvo lugar después de valorar las diferentes alternativas posibles que hasta entonces se habían encontrado, siempre pensando en el desarrollo sobre plataformas móviles. El proceso de elección del proyecto fue relativamente sencillo, pues la organización del departamento para esta finalidad es muy buena. Bastó con acudir al listado de proyectos ofertados, disponible online en [60], y consultar las diferentes posibilidades. Finalmente, el autor se puso en contacto con el tutor cuyo proyecto más interesaba y sin mayor problema quedó establecido el proyecto de fin de carrera.

Android era una plataforma relativamente desconocida para el autor, al menos cuando se inició el proyecto, pero no lo era sin embargo el crecimiento en importancia de las aplicaciones móviles. Por supuesto, la compañía responsable del sistema, Google, suponía un aliciente más a la hora de emprender esta aventura. Por tanto, se pueden resumir los motivos que llevaron al autor a decantarse por este proyecto:

- Mercado de dispositivos móviles muy interesante y en pleno crecimiento.
- Android, sistema novedoso y, a priori, fácil de aprender para el usuario por la utilización del lenguaje JAVA.
- Sistema impulsado por Google, compañía líder en el sector.

La duración en el tiempo del proyecto aproximadamente ha sido de **doce meses**. Durante este periodo de tiempo, el desarrollo del PFC se ha compaginado con otras actividades, por lo que la distribución del tiempo no ha sido uniforme. Se estima que el tiempo de trabajo real invertido en todo su desarrollo ha sido de **aproximadamente 705 horas**.

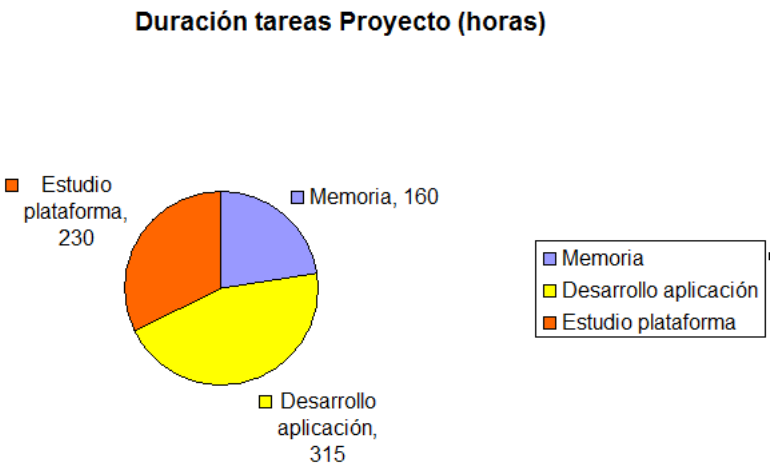

Figura 6.1: Duración de las tareas del proyecto.

Son muchas las tareas que lleva consigo realizar un proyecto de esta envergadura, aunque éstas se pueden englobar en tres grandes bloques:

- **Estudio de la plataforma Android**: aunque el lenguaje sobre el que se desarrolla la aplicación es JAVA y, por tanto, ya era conocido por el autor, ha sido necesaria la lectura de la documentación de Android, comprender el sistema operativo, documentación sobre las herramientas de desarrollo, instalación de SDK, etc. Este apartado está estrechamente relacionado con el desarrollo en sí. **230 horas**.

- **Desarrollo de la aplicación**: es el bloque al que se le han dedicado más horas de trabajo. Abarca todas las tareas relacionadas con la programación de la aplicación. Análisis, diseño, implementación y pruebas. **315 horas**.

- **Realización de la memoria**: la documentación abarca las tareas relacionadas con la redacción completa de la presente memoria, donde se incluye la descripción de Android y la aplicación desarrollada. Esta actividad tuvo una duración de **160 horas**.

En las figuras 6.1 y 6.2 se puede ver con más detalle el tiempo dedicado a la realización del proyecto, tanto los tiempos totales como divididos por tareas y meses.

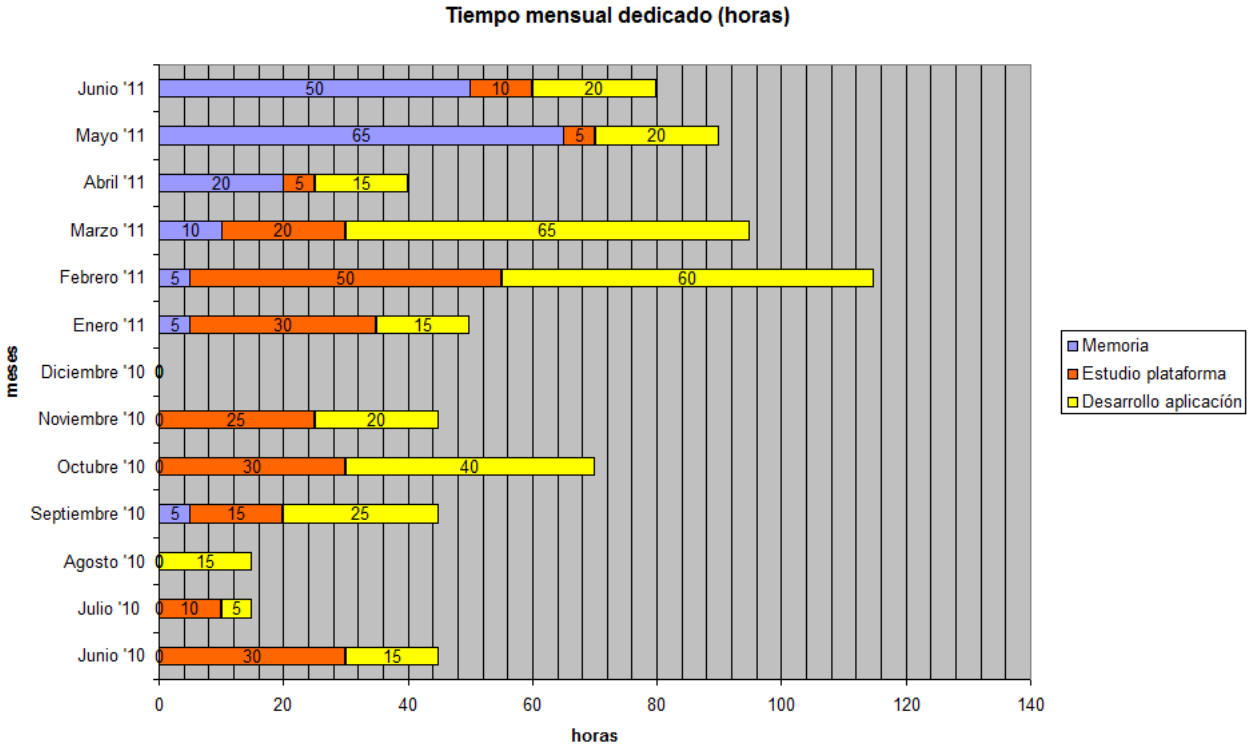

Figura 6.2: Tiempo dedicado por meses y tareas.

PERSONAL					
	Estudio (h)	Desarrollo (h)	Documentación (h)	Salario (€/h)	Euros
Analista Funcional	230	0	98	13,65	4.477,2
Analista Programador	0	315	62	15,85	5.975,45
TOTAL					10.452,65

Figura 6.3: Presupuesto para personal.

6.1. Presupuesto

A continuación se detalla el presupuesto final del proyecto. Los costes que conllevaf el desarrollo del presente proyecto se han dividido de la siguiente forma:

1. **Personal**
2. **Material e infraestructuras**

6.1.1. Personal

Para el desarrollo del proyecto se han utilizado profesionales de dos perfiles distintos pero complementarios, un analista funcional y un analista/programador. Debido al carácter académico del proyecto, éste ha sido llevado a cabo por una única persona que ha hecho las veces de analista y programador.

- **Analista funcional**: encargado de estudiar Android, diseñar la aplicación y redactar la documentación más relevante, representada por dos tercios de la presente memoria.
- **Analista programador**: encargado de la implementación de la aplicación y la documentación relativa a la misma.

Para hacer el cálculo, se ha obtenido el salario medio para esos dos perfiles en España en los últimos 12 meses [61]. En la figura 6.3 se refleja el número de horas que ha empleado cada uno de estos perfiles en el desarrollo del proyecto.

MATERIAL			
	Coste	Unidades	Total
Portátil HP Pavilion dv6-1250ss	680	1	680
Samsung Galaxy S	449	1	449
ADSL Movistar	39.99	13	519,87
TOTAL			1.648,87

Figura 6.4: Presupuesto para material.

6.1.2. Material e infraestructuras

Para el desarrollo del proyecto ha sido necesario el uso de material, tanto hardware como software, así como una conexión a Internet, imprescindible para poder realizar las pruebas funcionales de la aplicación. A continuación se detalla el material utilizado así como sus características:

- **Ordenador portátil HP Pavilion dv6-1250ss**, con procesador Intel Core Duo P7350 a 2GHz, memoria RAM DDR de 4 GB, disco duro de 500 GB de capacidad entre otras características.
- **Teléfono móvil Samsung Galaxy S**: pantalla Super AMOLED de 4,0 pulgadas, procesador de 1 GHz y 16 GB de memoria interna entre otras cualidades.
- **Conexión ADSL**: ADSL Movistar a 6MB de velocidad.

En la figura 6.4 se muestra el desglose del coste del material.

6.1.3. Resumen de costes

Resumiendo lo visto en las anteriores secciones acerca del presupuesto, el total del mismo se obtendrá como el resultado de la suma de los costes de personal y material utilizado. El **presupuesto total** de este proyecto asciende a la cantidad de **DOCE MIL CIENTO UN EUROS CON CINCUENTA Y DOS CÉNTIMOS**. En la figura 6.5 se muestra la tabla resumen del presupuesto.

RESUMEN DE COSTES	
Tipo de gasto	**Euros**
Gastos de personal	10.452,65
Gasotos de material	1.648,87
TOTAL	**12.101,52**

Figura 6.5: Presupuesto total del proyecto.

Capítulo 7

Conclusiones y trabajos futuros

En este capítulo se hace un repaso global al proyecto, presentado las conclusiones finales en comparación con los objetivos marcados inicialmente y los posibles trabajos futuros.

7.1. Conclusiones finales

Una vez concluidas las principales tareas que forman este proyecto fin de carrera, es el momento en el que se puede hacer balance y crítica de los resultados obtenidos. Así pues, repasando los objetivos inicialmente marcados pueden sacarse las siguientes conclusiones:

- **Familiarización con tecnologías Podcast**: aunque pueda no parecerlo, el término podcasts es relativamente desconocido para mucha gente. A pesar de que la popularidad de la red ha ido creciendo exponencialmente a lo largo de estos últimos años, es mucha la gente que desconoce la existencia de esta forma de descargar y escuchar sus programas favoritos. A lo largo del desarrollo del proyecto se ha hablado con asiduidad sobre podcasts, *feeds*, *RSS*. El desarrollo del proyecto ha servido sin duda como plataforma para familiarizarse y sacar provecho de esta forma de comunicación que son los podcasts. No sólo se ha hablado de sus ventajas para el usuario final, sino que también se ha profundizado en su funcionamiento interno para poder integrar esta tecnología en la aplicación desarrollada.

- **Técnicas de desarrollo con Android**: Android ofrece muchísimas posibilidades al desarrollador a la hora de crear una aplicación. Mediante una gran variedad de APIs proporciona al desarrollador acceso a las funcionalidades y servicios más comunes de los teléfonos móviles

actuales como acceso a Internet, servicios de reproducción de audio, bluetooth, servicio de localización, telefonía, etc. La aplicación implementada hace uso de varias funcionalidades interesantes, por lo que puede servir de referencia para el desarrollo de otras muchas aplicaciones. Además, para hacerla más interesante a nivel visual, se ha hecho uso de diversos efectos gráficos que Android pone a disposición de los programadores.

Se han incluido ejemplos de código de las partes más relevantes de la aplicación, de modo que cualquier otro desarrollador pudiera acudir al manual como referencia. Algunos casos de funcionalidades ejemplificadas son:

- Diseño y creación de interfaz de usuario con archivos XML.
- Manipulación y acceso a los elementos de la interfaz de usuario.
- Manejo de la BBDD.
- Creación y acceso a los Servicios de Audio, Descarga y Actualización.
- Manejo de hilos para operaciones en paralelo.
- Realización de peticiones HTTP.
- Procesamiento de archivos XML procedentes de la Red.

- **Diseñar, implementar y documentar una aplicación Android**: las conclusiones finales de desarrollar una aplicación para Android, desde la idea original, pasando por el diseño, implementación, batería de pruebas y documentación ha resultado bastante positiva. El hecho de que el lenguaje mayormente utilizado para el desarrollo sea JAVA junto con XML, hace que desde un principio uno se encuentre cómodo aprendiendo a utilizar la plataforma. Ademas, los numerosos tutoriales de iniciación incluidos en la página oficial de Android así como la extensa comunidad de desarrolladores que desde un principio ha existido para la plataforma ayudan a que el desarrollador tenga siempre una fuente muy fiable a donde acudir a buscar información o ayuda para el desarrollo de la aplicación.

  Android es, por tanto, una plataforma de lo más interesante para que un desarrollador realice sus ideas. No sólo es atractiva a nivel de mercado, sino que además intenta facilitar al desarrollador el máximo posible su tarea.

7.2. Dificultades

Android es una tecnología aún muy reciente, lo que implica algunas dificultades a la hora de crear aplicaciones sobre la misma

- **Documentación**: aunque es cierto que casi todos los elementos están bien documentados por parte de la propia Google, existen todavía ciertos aspectos que o bien aún no están documentados, o la documentación existente es muy escasa. Conforme ha ido pasando el tiempo el problema es cada vez menor, dado que Google va mejorando y actualizando constantemente la documentación disponible, pero sí que suponía un problema sobre todo al principio del desarrollo del proyecto.

- **Actualizaciones de Android**: al igual que ocurre con la documentación, este es un problema que ha ido menguando con el paso del tiempo. Desde el principio, Google ha ido desarrollando y perfeccionando el sistema Android de una manera constante. Esto se ha ido traduciendo en actualizaciones de la plataforma con soluciones a bugs más o menos importantes, creación de nuevas herramientas de desarrollo, incorporación de más y mejores componentes para las aplicaciones, etc. Aunque aparentemente esto pueda ser bueno, también trae consigo algunas consecuencias negativas. Por ejemplo, problemas de incompatibilidades entre versiones de Android, por lo que a veces el desarrollador se puede ver obligado a modificar partes de código que teóricamente ya estaban funcionando. En cualquier caso, Google siempre pone a disposición de los desarrolladores la documentación acerca de los cambios de cada versión [62].

- **Bugs existentes en la plataforma**: una vez más, la juventud de la plataforma hace que también la existencia de *bugs* sea relativamente frecuente. Durante el desarrollo de la aplicación se ha hecho frente a algunos de estos problemas, teniendo que hacer desarrollos alternativos en alguna ocasión. Google pone a disposición de los usuarios una página donde denunciar estos bugs [63] y donde se ofrece información acerca de ellos.

7.3. Trabajos futuros

Las aplicaciones de este tipo siempre son susceptibles de futuras ampliaciones o mejoras. Muchas veces, bien por falta de tiempo o bien por falta de medios, existen ideas interesantes sobre las que mejorar la aplicación pero

que finalmente no pueden incluirse en el desarrollo. El caso de DroidCatcher no es una excepción.

La plataforma Android es todavía muy joven y, por tanto, está en continua evolución. Google ha diseñado un programa de actualización muy agresivo desde el comienzo de la plataforma, con importantes mejoras en el software y actualizaciones constantes en las librerías de desarrollo. A lo largo del desarrollo de la aplicación se han tenido en cuenta algunas de estas mejoras que han ido surgiendo pero otras, en cambio, no han podido aplicarse por las dificultades que hubieran supuesto.

- **Mejoras en el Gestor de Descargas**: Google ha incluido en las actualizaciones más recientes de la plataforma nuevas librerías para gestionar de manera más eficiente la descarga de archivos de Internet. Con estás nuevas librerías, se podría replantear la funcionalidad Gestor de Descargas de la aplicación para incluir una serie de mejoras en ella. Cosas como descarga simultánea de múltiples ficheros (debido también a la incorporación de nuevos y mas potentes procesadores de doble núcleo), poder pausar y reanudar descargas, etc.

- **Mejoras en la implementación de los Servicios**: la mayoría de las aplicaciones no deberían utilizar AIDL para crear un servicio, debido a que puede requerir capacidades multi-hilo y puede resultar en una implementación más complicada. Debido al carácter académico de la aplicación, se decidió utilizar AIDL para la creación del servicio, pero no por ello ha de ser la mejor opción. Una futura revisión podría contemplar una revisión de esta implementación, valorando las distintas posibilidades que ofrece la plataforma [64] en función de las necesidades reales de la aplicación.

Apéndice A

Manual de instalación y desarrollo

Desarrollar aplicaciones para Android es posible gracias a un grupo de herramientas que proporciona el SDK. Se puede acceder a estas herramientas a través del *plugin* de Elclipse, ADT, o a través de la línea de comandos. El método predilecto para desarrollar las aplicaciones es utilizando Eclipse, pues desde ahí se podrán invocar las herramientas necesarias para el desarrollo de la aplicación.

A continuación se explican los pasos necesarios para la instalación de DroidCatcher, así como de los pasos necesarios para desarrollar una aplicación desde el principio utilizando Eclipse, pues es la plataforma que se ha utilizado para el desarrollo del PFC.

A.1. Instalación de DroidCatcher

El único fichero necesario para la instalación de una aplicación Android es el archivo .apk, resultante de la compilación de dicha aplicación. Dentro del proyecto para Eclipse llamado DroidCatcher, presente en el CD adjunto, se podrá encontrar el fichero de nombre 'DroidCatcher.apk'.

Para instalar la aplicación directamente en el dispositivo es necesario indicar explícitamente en el teléfono que se permite la instalación de aplicaciones fuera del servicio Android Market de Google. Para ello:

Ajustes → Aplicaciones → Fuentes desconocidas

Finalmente, bastará con copiar el archivo .apk en la memoria del teléfono, acceder a él a través del explorador de archivos y pulsar sobre él para iniciar la instalación.

A.2. Instalación del SDK

Antes de empezar con la instalación de la plataforma, es necesario comprobar que se cumplen los requisitos mínimos del sistema [65]. Una vez confirmado este punto, es necesario descargar e instalar el SDK de Android.

A.2.1. SDK de Android

En la página de descargas de Android [62] se puede obtener la última versión del SDK (versión 3.1 a fecha de Mayo de 2011), en versión ejecutable o comprimida en un fichero *.zip*. Una vez obtenido el SDK, bastará con descomprimirlo en la ruta que el usuario prefiera. Esta ruta será utilizada posteriormente para hacer referencia al SDK, por lo que será necesario recordarla.

A.2.2. Instalación de ADT

ADT es un plugin para el entorno de desarrollo Eclipse. Extiende las capacidades de Eclipse de modo que, utilizando las herramientas proporcionadas por el SDK de Android, permite configurar rápidamente proyectos, crear interfaces de usuario y depurar las aplicaciones Android.

Antes de poder instalar ADT, es necesario tener instalada una versión compatible de Eclipse. Si no es el caso, se puede descargar desde la página oficial de descargas de Android, [66].

Descargar plugin ADT

A continuación se detallan los pasos necesarios para instalar el plugin ADT en el entorno Eclipse:

1. Iniciar Eclipse, seleccionar **Help** → **Install New Software...**
2. Pinchar en **Add**, en la esquina superior derecha.
3. En el diálogo que aparece (*Add repository*), introducir "ADT Plugin" como nombre y la siguiente URL en el campo **Location**: *https://dl-ssl.google.com/android/eclipse/*
4. Click en OK
5. En el diálogo de **Available Software**, marcar la casilla al lado de **Developer Tools** y hacer pulsar siguiente.

6. En la siguiente ventana, se verá una lista con las herramientas que serán descargadas. Pulsar en siguiente.

7. Leer y aceptar los términos de la licencia y finalmente pinchar en **Finish**.

8. Reiniciar Eclipse cuando se haya completado la instalación.

Configuración de ADT

Después de descargar e instalar correctamente el plugin ADT, el siguiente paso es configurarlo para que apunte al directorio donde se encuentra la instalación del SDK de Android:

1. Seleccionar **Window** → **Preferences**.

2. Seleccionar **Android** en el panel de la izquierda.

3. En el panel principal, en **SDK Location**, pinchar en **Browse...** y seleccionar el directorio donde se instaló el SDK de Android.

4. Pinchar en **Apply**, después en **OK**.

Si todo ha ido correctamente, la instalación del plugin ya estaría terminada.

A.3. Configuración del Dispositivo Virtual Android, AVD

El *AVD Manager* es una interfaz de usuario fácil de manejar que permite gestionar las distintas configuraciones de Dispositivos Virtuales de Android, **AVD** en sus siglas en inglés. Un AVD es una configuración del emulador de Android que permite modelar diferentes características de dispositivos que utilizan dicho sistema operativo.

Se pueden crear tantos AVDs como se quieran, de modo que se puedan realizar pruebas de las aplicaciones sobre dispositivos (virtuales) con distintas configuraciones.

Para crear un AVD:

1. Iniciar AVD Manager desde eclipse. Para ello, seleccionar **Window** → **Android SDK and AVD Manager** o pinchar en el icono de AVD Manager en la barra de herramientas de Eclipse.

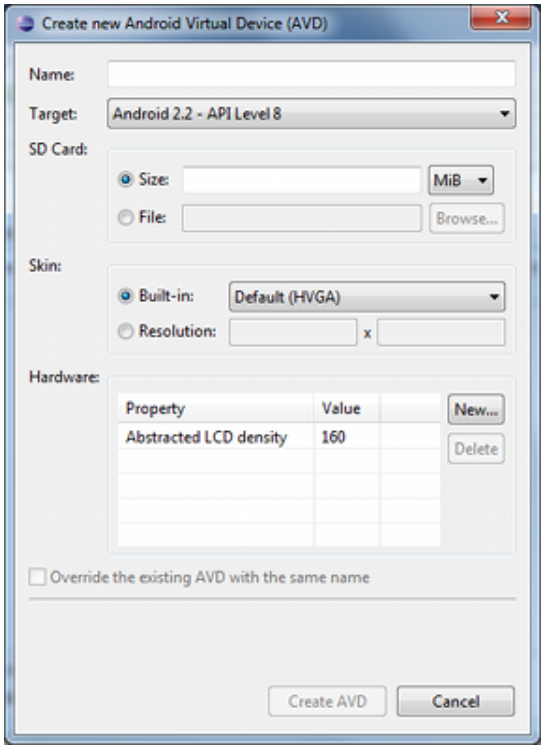

Figura A.1: Creación de AVD

2. En el panel *Viertual devices*, aparecerá un listado con los AVDs dispoibles. Para crear uno nuevo, pinchar en **New**. Aparecerá el diálogo **Create New AVD**.

3. Rellenar los campos con los datos del nuevo AVD. Nombre, plataforma, tamaño de la memoria de la tarjeta SD y apariencia. Además, se pueden especificar más características de hardware pinchando en el botón **New...**. Se puede ver un listado completo de las características de hardware en [67].

4. Finalmente, pinchar en **Create AVD**.

A.4. Creación de un proyecto Android

El plugin ADT proporciona un diágolo de Ñuevo Proyecto'que se puede utilizar para crear proyectos Android de forma rápida y sencilla. Para crearlo:

1. Seleccionar **File** → **New** → **Project**.

2. Seleccionar **Android** → **Android Project** y pinchar en **Next**.

3. Selccionar los contenidos del proyecto:

   - Introducir el nombre del proyecto. Éste será el nombre de la carpeta donde se cree el proyecto.
   - Debajo de **Contents**, seleccionar **Create new project in workspace**. Establecer la ubicación del proyecto.
   - Debajo de **Target**, seleccionar un **Android target**, que será utilizado como nivel de compilación del proyecto. Esto indica que plataforma Android se utilizará.
   - En **Properties**, rellenar los campos necesarios:
     - **Application name**. Título *legible* de la aplicación. El nombre que aparecerá en el dispositivo del usuario.
     - **Package name**. Espacio de nombres para la aplicación. Sigue la nomenclatura de los paquetes de java.
     - **Min. sdk version**. Entero que indica el nivel mínimo de API requerido para que la aplicación se ejecute correctamente.

4. **Finish**

A.5. Compilación y ejecución de la aplicación

Eclipse, junto con el plugin ADT, proporcionan un entorno de desarrollo donde la mayor parte de los detalles del proceso de compilación son transparentes al desarrollador. Por defecto, el proceso de compilación se está ejecutando continuamente en segundo plano mientras que el hace cambios en el proyecto.

Cuando Eclipse compila automáticamente la aplicación, habilita la depuración y firma el *.apk* con una clave por defecto. Cuando se ejecuta la aplicación, Eclipse invoca ADB e instala la aplicación en un dispositivo o emulador, de manera que el desarrollador no tenga que hacer estas tareas manualmente. A continuación se explican los pasos para ejecutar una aplicación utilizando Eclipse, tanto en un emulador con en un dispositivo real.

A.5.1. Ejecución en un emulador

Para ejecutar la aplicación, seleccionar **Run** → **Run** (o **Run** → **debug**) desde el menú de Eclipse. El plugin ADT creará automáticamente una

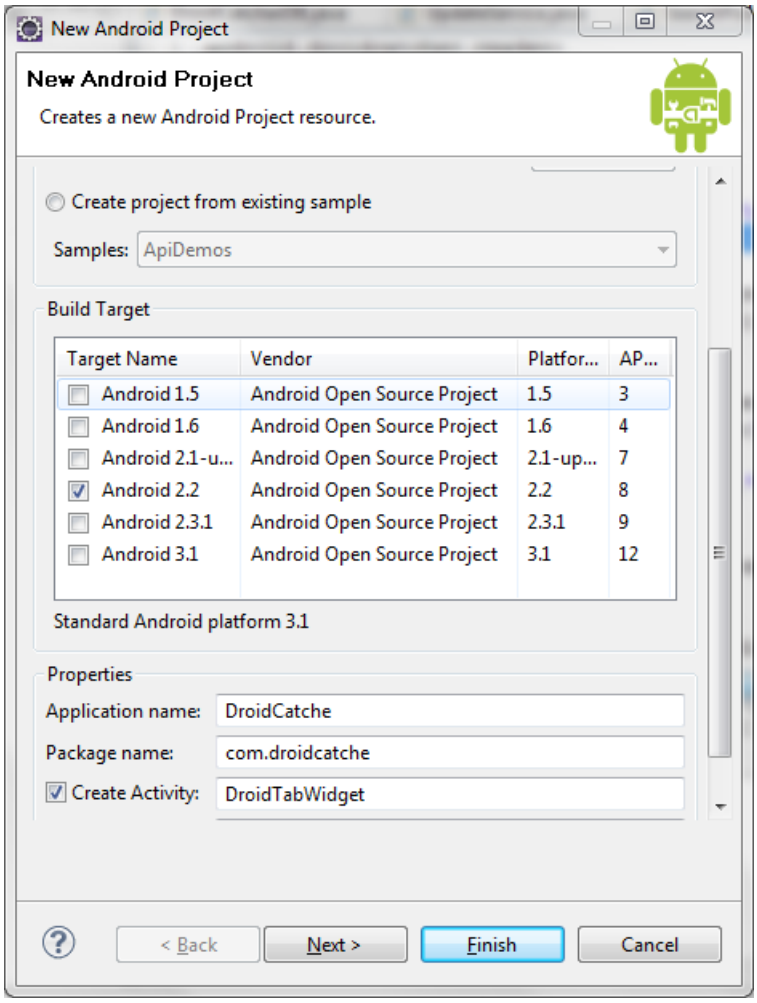

Figura A.2: Creación de un nuevo proyecto en Eclipse

configuración de ejecución por defecto para el proyecto. Eclipse entonces realizará las siguientes tareas:

1. Compilar el proyecto (si ha habido cambios desde la última compilación).
2. Crear una configuración de ejecución por defecto (si no existe ya una para el proyecto).
3. Instalar e iniciar la aplicación en el emulador, basado en *Deployment Target* definido en la configuración de ejecución.

Si se ejecuta la aplicación en modo *Debug*, se iniciará en modo *Waiting for Debugger*. Una vez que el depurador se haya unido, Eclipse abrirá la perspectiva de depuración e iniciará la Activity principal de la aplicación. De otra forma, si se ejecuta la aplicación en modo normal, Eclipse instala la aplicación en el dispositivo e inicia la Activity principal de la aplicación.

A.5.2. Ejecución en un dispositivo

Antes de poder ejecutar la aplicación en un dispositivo, se deben realizar ciertas tareas de configuración para el mismo:

- Hay que asegurarse de que se puede depurar la aplicación estableciendo a `true` el atributo `android:debugable` de el elemento <`application`>.
- Habilitar *USB Debugging* en el dispositivo. En la mayoría de los dispositivos, esta configuración se encuentra en **Settings** → **Applications** → **Development** → **USB Debugging**

Una vez configurados estos parámetros y estando el disposivito conectado vía USB, instalar la aplicación en el dispositivo seleccionando **Run** → **Run** (o **Run** → **debug**) desde el menú de Eclipse

A.6. Depuración de la aplicación

Si se desarrolla la aplicación utilizando Eclipse junto con el plug-in ADT, entonces se puede hacer uso también del depurador de JAVA junto con DDMS para depurar las aplicaciones.

DDMS (Dalvik Debgug Monitor Server en inglés) [68], es una herramienta incluida en Android y que proporciona utilidades al usuario tales como

captura de pantalla, información de los hilos ejecutando en el dispositivo, información de trazas, procesos y estados, simulación de llamadas y SMS entrantes y muchas más cosas.

Para acceder al depurador de JAVA y a DDMD, Eclipse muestra ambos como perspectivas, que son vistas adaptadas para a Eclipse donde se muestran diversas pestañas y ventanas dependiendo de en qué perspectiva se encuentre uno. Eclipse también se encarga de iniciar al proceso ADB automáticamente, de modo que el desarrollador no se tenga que preocupar por ello.

A.6.1. La perspectiva `Debug` en Eclipse

La perspectiva para la depuración en Eclipse, Debug, da acceso a las siguientes pestañas:

- **Debug**: muestra las aplicaciones que han sido previamente depuradas o que se están depurando en este momento y los hilos que se están ejecutando actualmente.
- **Variables**: cuando se establecen puntos de parada (breakpoints), muestra los valores de las variables durante la ejecución del código.
- **Breakpoints**: muestra un listado con los puntos de parada que se han establecido en el código de la aplicación.
- **LogCat**: permite ver mensajes de trazas del sistema en tiempo real. La pestaña LogCat también está disponible en la perspectiva DDMS.

Se puede acceder a esta perspectiva pinchando en **Window** → **Open Perspective** → **Debug**. Para más información, referirse a la documentación del depurador de Eclipse [69].

A.6.2. La perspectiva `DDMS` en Eclipse

Esta perspectiva de Eclipse permite al desarrollador acceder a todas las características de DDMS desde dentro del entorno de desarrollo Eclipse. En la figura A.3 se puede observar las secciones de DDMS que están disponibles:

- **Devices**: muestra un listado de los dispositivos y ADVs que están conectados con ADB.
- **Emulator Control**: permite llevar a cabo las funciones del dispositivo.
- **LogCat**: permite ver mensajes de trazas del sistema en tiempo real.

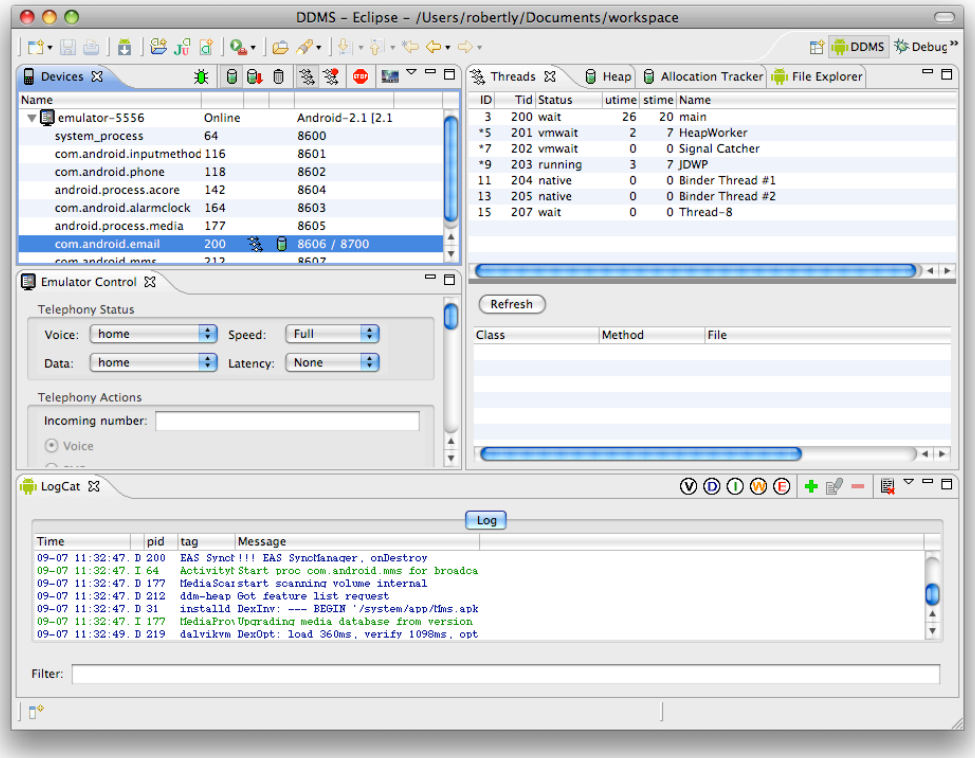

Figura A.3: Perspectiva DDMS en Eclipse

- **Threads**: muestra los hilos que se están ejecutando actualmente dentro de la máquina virtual.
- **Heap**: muestra el uso de la pila por parte de la máquina virtual.
- **Allocation Tracker**: muestra la asignación de memoria de los objetos.
- **File Explorer**: permite explorar el sistema de archivos del dispositivo.

Para acceder a la perspectica DDMS, **Window** → **Open Perspective** → **DDMS**. Si no aparece DDMS, entonces hacer **Window** → **Open Perspective** → **Other...** y seleccionar DDMS en la ventana que aparece. Para más información acerca de DDMS, referirse a la documentación oficial de utilización de DDMS [68]

Apéndice B

Glosario

- **AIDL**: siglas de Android Interface Definition Language, Lenguaje para la Definición de Interfaces en Android. Constituye un lenguaje de sintaxis muy básica que permite describir interfaces que pueden ser utilizadas de forma remota.
- **API**: siglas de Application Programming Interface, Interfaz de Programación de Aplicaciones. Consiste en un conjunto de llamadas que ofrecen acceso a funciones y procedimientos, representando una capa de abstracción para el desarrollador.
- **Bytecode**: código intermedio, más abstracto que el código máquina, y que necesita de un mediador o máquina virtual para poder ser transformado y ejecutado en un hardware local.
- **Callback**: se denomina así a la relación que existe entre dos procesos cuando el origen de la comunicación es a su vez llamado o invocado por el proceso destino.
- **Checkbox**: elemento de interfaz de usuario que permite hacer una selección múltiple en un conjunto de opciones.
- **Dalvik**: nombre de la máquina virtual utilizada por el sistema operativo Android. Dalvik esta específicamente adaptada a las características de rendimiento de un dispositivo móvil y trabaja con ficheros de extensión ".dex", obtenidos desde el bytecode de Java.
- **EDGE**: siglas en inglés de Enhanced Data rates for GSM of Evolution, Tasas de Datos Mejoradas para la evolución de GSM. Es una tecnología para telefonía móvil que representa un puente entre la segunda y tercera generación de estos dispositivos.

- **Episodio**: cualquier fichero dentro feed RSS de un Podcast. Se les llama episodios porque normalmente son archivos de audio o vídeo, pero cualquier fichero de un Podcast podría ser considerado un episodio. Muchos podcasts están limitados únicamente a documentos PDF. Cada uno de estos documentos sería considerado un episodio. De forma similar, si un Podcast contiene tanto archivos de audio como de vídeo, cada uno de ellos sería considerado un episodio individual incluso si estuvieran relacionados entre ellos.

- **Framework**: término con el que se define un amplio conjunto de elementos que permite desarrollar y organizar software utilizando un determinado lenguaje, sistema o tecnología. Habitualmente incluye bibliotecas, programas de desarrollo o manuales.

- **GPRS**: siglas de General Packet Radio Service, Servicio General de Paquetes vía Radio. es una extensión del estándar GSM que permite mejorar sus prestaciones originales, como el envío de datos, uso de correo electrónico, navegación web o el aumento de las tasas de transferencia de datos.

- **GSM**: siglas de Groupe Spécial Mobile, más conocido como Sistema Global para las Comunicaciones Móviles, es el estándar más extendido para las comunicaciones con telefonía móvil. Permite llamadas, navegación por Internet o envío de SMS.

- **GUI**: siglas de Graphical User Interface, Interfaz Gráfica de Usuario. Representa la parte del software que, mediante un contexto o lenguaje principalmente visual y simbólico, permite al usuario utilizar una aplicación.

- **Hilo**: en sistemas operativos, un hilo constituye cada uno de los flujos de ejecución en el que puede ser divido un proceso. Todos los hilos de un proceso comparten espacio en memoria, archivos abiertos, variables globales, semáforos, etc. Permiten la ejecución concurrente de varias tareas. También llamado *thread*.

- **HTTP**: siglas de HyperText Transfer Protocol, Protocolo de Transferencia de Hipertexto. Constituye el protocolo utilizado para la transmisión de documentos a través de la Web entre un cliente y servidor.

- **JAR**: acrónimo de Java ARchive, en español Archivo Java. Representa una agrupación de varios ficheros Java y se usa generalmente para la distribución conjunta de clases y metadatos.

- **JVM**: siglas de Java Virtual Machine, en español Máquina Virtual de Java. Constituye un elemento software de la tecnología Java, encargado de transformar el código intermedio universal o bytecode en código máquina específico del hardware donde está instalado.
- **Kernel**: parte fundamental de un sistema operativo, responsable de facilitar acceso seguro al hardware, gestionar recursos y hacer llamadas al sistema. También conocido como núcleo.
- **Listener**: objeto que está a la espera de determinado evento.
- **Máquina virtual**: representa un software que emula el comportamiento de una determinada arquitectura o que permite adaptar un código fuente a las características de la máquina nativa.
- **MP3**: formato de compresión de audio digital cuyo nombre completo es MPEG-1 Audio Layer 3. También utilizado, por extensión, para nombrar al dispositivo móvil reproductor de audio digital en el que se almacena, organiza o reproduce archivos de audio digital.
- **PDA**: siglas de Personal Digital Assistant, en español Asistente Digital Personal. Dispositivo móvil utilizado como organizador personal, que cuenta generalmente con pantalla táctil, agenda, calendario, conectividad Wi-Fi, y aplicaciones ofimáticas, entre otros.
- **Plug-in**: componente de software que se relaciona y ejecuta con otro para aportarle una función nueva y generalmente muy especifica.
- **Podcast**: una serie de ficheros agrupados por un 'Podcaster'y distribuidos por Internet mediante un feed RSS al cual se pueden suscribir usuarios utilizando un 'Podcatcher'.
- **Podcaster**: cualquiera que publique un Podcast. Muchos podcasts son publicados por un grupo de Podcasters o incluso por corporaciones dedicadas a su producción.
- **Podcatcher**: cualquier programa que comprueba regularmente un feed RSS en busca de actualizaciones de un Podcast. Éstos podrán o no podrán ser capaces de poner estos ficheros dentro de un iPod o cualquier otro dispositivo multimedia portátil. Los podcatchers más comunes incluyen iTunes de Apple y un programa de código abierto llamado Juice.

- **Proceso**: un proceso es un programa en ejecución, y representa la unidad de procesamiento básica gestionada por el sistema operativo.
- **RMI**: siglas de Remote Method Invocation, en español Invocación de Métodos Remotos, es una tecnología de Java que permite comunicar objetos distribuidos escritos en este lenguaje.
- **RSS** siglas de Really Simple Syndication, un formato XML para sindicar o compartir contenido en la web. Se utiliza para difundir información actualizada frecuentemente a usuarios que se han suscrito a la fuente de contenidos. El formato permite distribuir contenidos sin necesidad de un navegador, utilizando un software diseñado para leer estos contenidos RSS (agregador)
- **SAX**: siglas de Simple API for XML, en español API Simple para XML, representa una conocida API para Java que facilita el procesamiento de documentos XML.
- **Sistema operativo**: programa cuya finalidad principal es simplificar el manejo y explotación de un elemento con capacidad computacional, gestionando sus recursos, ofreciendo servicios a las demás aplicaciones y ejecutando mandatos del usuario.
- **SDK**: siglas de Software Development Kit, en español Kit de Desarrollo de Software. Constituye un conjunto de herramientas que permiten a un desarrollador crear aplicaciones para una determinada plataforma o lenguaje.
- **SmartPhone**: es un teléfono inteligente que puede comunicarse a través de Wi-Fi, bluetooth, conexión al Internet, envío de mensajería, e-mails. Generalmente se define como dispositivo electrónico de mano que integra la funcionalidad de un teléfono celular, PDA o similar. Generalmente se realiza añadiendo funciones de teléfono a un PDA existente o añadiendo funcionalidades ïnteligentes", como las funciones del PDA, en un teléfono celular. Una característica clave de un smartphone es que las aplicaciones adicionales pueden ser instaladas en el dispositivo. Las aplicaciones puede ser desarrolladas por el fabricante del dispositivo, por el operador o por cualquier empresa derrolladora de software.
- **UMTS**: siglas de Universal Mobile Telecommunications System, en español Sistema Universal de Telecomunicaciones Móviles. Constituye en estándar de comunicación para dispositivos de tercera generación o

3G, que ofrece capacidades multimedia y conexiones de alta velocidad en Internet.

- **URL**: siglas en inglés de Uniform Resource Locator, Localizador Uniforme de Recursos, es una secuencia de caracteres, de acuerdo a un formato modélico y estándar, que se usa para nombrar recursos en Internet para su localización o identificación, como por ejemplo documentos textuales, imágenes, vídeos, presentaciones digitales, etc.

- **Wi-Fi**: acrónimo de Wíreless Fidelity, estándar de envío de datos que utiliza ondas de radio en lugar de cables.

- **WAP**: siglas de Wireless Application Protocol, en español o Protocolo de Aplicaciones Inalámbricas. Es un estándar para aplicaciones que utilizan las comunicaciones inalámbricas, como el acceso a servicios de Internet desde un teléfono móvil.

- **Widget**: componente gráfico utilizado en interfaces de usuario, con el cual el usuario puede interactuar, como por ejemplo cajas de texto, botones, ventanas, etc.

- **XML**: siglas de Extensible Markup Language, en español Lenguaje de Marcado Extensible. Representa un lenguaje estándar que, mediante el uso de etiquetas y atributos, permite expresar e intercambiar fácilmente estructuras de datos.

- **XMPP**: siglas de Extensible Messaging and Presence Protocol, en español Protocolo Extensible de Mensajería y Presencia. Es un protocolo basado en XML que se utiliza en servicios de mensajería instantánea.

- **3G**: es la abreviación de tercera generación de transmisión de voz y datos a través de telefonía móvil. La definición técnicamente correcta es UMTS (Universal Mobile Telecommunications System o Servicio Universal de Telecomunicaciones Móviles)

Bibliografía

[1] About.com. Definición de smartphone. http://cellphones.about.com/od/glossary/g/smart_defined.htm.

[2] M. Jukov. Mobile applications - the history of the issue. http://www.articledashboard.com/Article/Mobile-applications---the-history-of-the-issue/1048768.

[3] Wikipedia.org. Wap. http://es.wikipedia.org/wiki/WAP.

[4] Wikipedia.org. Gprs. http://es.wikipedia.org/wiki/GPRS.

[5] Wikipedia.org. Edge. http://es.wikipedia.org/wiki/EDGE.

[6] Wikipedia.org. Telefonía 3g. http://es.wikipedia.org/wiki/3G.

[7] Wikipedia.org. Windows mobile. http://es.wikipedia.org/wiki/Windows_Phone.

[8] Symbian Foundation. Fundación symbian. http://licensing.symbian.org/.

[9] Research in Motion. Research in motion. http://www.rim.com/.

[10] Apple Corp. Tecnología ios. http://developer.apple.com/technologies/ios/.

[11] Online MBA. Evolución de aplicaciones móviles. http://www.onlinemba.com/blog/apps-apps-apps/.

[12] Google Inc. Página web principal de android. http://developer.android.com/index.html.

[13] Google Inc. Fundamentos de android. http://developer.android.com/guide/basics/what-is-android.html.

[14] Google Inc. Android 2.0 platform highlights. http://developer.android.com/sdk/android-2.0-highlights.html.

[15] Adobe Systems Incorporated. Adobe flash availability. http://www.adobe.com/aboutadobe/pressroom/pressreleases/201006/06222010FlashPlayerAvailability.html.

[16] Google Inc. Android 2.2 platform highlights. http://developer.android.com/sdk/android-2.2-highlights.html.

[17] Google Inc. Android 2.3 platform highlights. http://developer.android.com/sdk/android-2.3-highlights.html.

[18] Google Inc. Android 3.0 platform highlights. http://developer.android.com/sdk/android-3.0-highlights.html.

[19] Engadget. Next version of android will combine gingerbread and honeycomb. http://www.engadget.com/2011/02/15/next-version-of-android-will-combine-gingerbread-and-honeycomb/.

[20] Google Inc. Dalvik vm internals. https://sites.google.com/site/io/dalvik-vm-internals.

[21] Google Inc. Dalvik jit. http://android-developers.blogspot.com/2010/05/dalvik-jit.html.

[22] Google Inc. Componentes de una aplicación. http://developer.android.com/guide/topics/fundamentals.html#Components.

[23] Google Inc. Androidmanifest. http://developer.android.com/guide/topics/manifest/manifest-intro.html.

[24] Google Inc. Api de android, activity. http://developer.android.com/reference/android/app/Activity.html.

[25] Google Inc. Fundamentos de android, activity. http://developer.android.com/guide/topics/fundamentals/activities.html.

[26] Google Inc. Fundamentos de android, service. http://developer.android.com/guide/topics/fundamentals/services.html.

[27] Google Inc. Api de android, service. http://developer.android.com/reference/android/app/Service.html.

[28] Google Inc. Api de android, ciclo de vida de proceso. http://developer.android.com/reference/android/app/Service.html#ProcessLifecycle.

[29] Google Inc. Creación de un servicio extendiendo ibinder. http://developer.android.com/guide/topics/fundamentals/bound-services.html#Binder.

[30] Google Inc. Creación de un servicio usando messenger. http://developer.android.com/guide/topics/fundamentals/bound-services.html#Messenger.

[31] Google Inc. Android interface definition language. http://developer.android.com/guide/developing/tools/aidl.html.

[32] Google Inc. Api de android, broadcastreceiver. http://developer.android.com/reference/android/content/BroadcastReceiver.html.

[33] Google Inc. Api de android, contentprovider. http://developer.android.com/guide/topics/providers/content-providers.html.

[34] Google Inc. Api de android. intent. http://developer.android.com/reference/android/content/Intent.html.

[35] Google Inc. Activando componentes. http://developer.android.com/guide/topics/fundamentals.html#ActivatingComponents.

[36] Google Inc. Resolución de intents. http://developer.android.com/guide/topics/intents/intents-filters.html#ires.

[37] Google Inc. Interfaz de usuario. http://developer.android.com/guide/topics/ui/index.html.

[38] Google Inc. Interfaz de usuario. layout. http://developer.android.com/reference/android/text/Layout.html.

[39] Google Inc. Interfaz de usuario. widgets. http://developer.android.com/guide/topics/ui/index.html#Widgets.

[40] Google Inc. Interfaz de usuario. menus. http://developer.android.com/guide/topics/ui/index.html#Menus.

[41] Google Inc. Interfaz de usuario. eventos del interfaz de usuario. http://developer.android.com/guide/topics/ui/index.html#Events.

[42] Google Inc. Almacenamiento de datos. http://developer.android.com/guide/topics/data/data-storage.html.

[43] Google Inc. Preferencias compartidas. http://developer.android.com/guide/topics/data/data-storage.html#pref.

[44] Google Inc. Almacenamiento interno. http://developer.android.com/guide/topics/data/data-storage.html#filesInternal.

[45] Google Inc. Almacenamiento externo. http://developer.android.com/guide/topics/data/data-storage.html#filesExternal.

[46] Google Inc. Base de datos, sqlite. http://developer.android.com/guide/topics/data/data-storage.html#db.

[47] Google Inc. Almacenamiento de datos, conexión de red. http://developer.android.com/guide/topics/data/data-storage.html#netw.

[48] World Wide Web Consortium. Especificación xml. http://www.w3.org/XML/.

[49] RSS Advisory Board. Especificación rss 2.0. http://www.rssboard.org/rss-specification.

[50] RSS Advisory Board. Especificación rss 2.0, etiqueta channel. http://www.rssboard.org/rss-specification#requiredChannelElements.

[51] RSS Advisory Board. Especificación rss 2.0, item. http://www.rssboard.org/rss-specification#hrelementsOfLtitemgt.

[52] Andev. Andev.org. http://www.anddev.org/.

[53] HelloAndroid. Helloandroid.com. http://www.helloandroid.com/.

[54] Stack Overflow. Stack overflow. http://stackoverflow.com/questions/tagged/android.

[55] Google Inc. Tutorial tab layout. http://developer.android.com/resources/tutorials/views/hello-tabwidget.html.

[56] The SQLite Consortium. Sitio web de sqlite. http://www.sqlite.org/.

[57] DigitalPodcast. Servicio de búsqueda de digitalpodcast. http://www.digitalpodcast.com/podcastsearchservice/.

[58] Podsonoro. Podsonoro.com. http://www.podsonoro.com/.

[59] Google Inc. Animaciones en android. http://developer.android.com/reference/android/view/animation/package-summary.html.

[60] UC3M. Listado pfc uc3m. http://www3.uc3m.es/web/pfc/Depto126.xml.

[61] Infojobs. Sueldo medio por puesto de trabajo. http://salarios.infojobs.net/.

[62] Google Inc. Página de descargas de android. http://developer.android.com/sdk/index.html.

[63] Google Inc. Listado de problemas abiertos de android. http://code.google.com/p/android/issues/list.

[64] Google Inc. Formas de crear un servicio. http://developer.android.com/guide/topics/fundamentals/bound-services.html.

[65] Google Inc. Requerimientos del sistema. http://developer.android.com/sdk/requirements.html.

[66] The Eclipse Foundation. Sitio de descargas de eclipse. http://www.eclipse.org/downloads/.

[67] Google Inc. Configuración harware avd. http://developer.android.com/guide/developing/devices/managing-avds.html#hardwareopts.

[68] Google Inc. Herramienta ddms. http://developer.android.com/guide/developing/debugging/ddms.html.

[69] Eclipse Foundation. Documentación de eclipse. http://help.eclipse.org/helios/index.jsp.